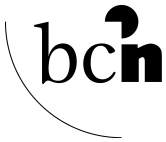
Using ILP to Learn Local Linguistic Structures

Stasinos Themistokleous Konstantopoulos



The work in this thesis has been carried out under the auspices of the Behavioral and Cognitive Neurosciences (BCN) research school, Groningen.

Groningen Dissertations in Linguistics 45
ISSN 0928-0030

Document prepared with LaTeX 2$_\varepsilon$ and typeset by pdfTeX,
Greek summary prepared with Lambda and typeset by Omega.

Rijks*universiteit* Groningen

Using ILP to Learn Local Linguistic Structures

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. F. Zwarts,
in het openbaar te verdedigen op
vrijdag 28 november 2003
om 14.15 uur

door

Stasinos Themistokleous Konstantopoulos

geboren op 2 september 1973
te Athene, Griekenland

Promotor:	Prof. dr. ir. J. Nerbonne
Beoordelingscommissie:	Prof. dr. W. Daelemans
	Prof. dr. T. Kuipers
	Prof. dr. G. R. Renardel de Lavalette

We in the West have no syllogism exactly equal to the anumana *and it is a shame that we do not, because had we such a rigorous form by which to check our inductive reasoning, Bishop Timothy Archer might well know of it, and had he known of it he would have known that his mistress waking up to find her hair singed does not, in fact, prove that the spirit of his dead son has returned from the other world.*

Philip K. Dick, The Transmigration of Timothy Archer

Preface

This thesis would not have been possible without the help of many people, to all of whome I'm deeply indebted. First of all I would like to thank John Nerbonne for supervising my project. Without his experience and guidance it would have taken a tougher and longer time, and the result would have been much less satisfactory. And particularly because without his tireless instigations to clearly and fully explain my thoughts on paper, this thesis would have been impossible to read.

Then I would like to thank all the people in Alfa-Informatica in particular and the H.1311 corridor in general for all the ideas we have exchanged and all the coffees we've had together. (I hope I got the corridor number right, one can never be sure with the Harnonie building's numbering scheme, but anyway you know which corridor I mean.) I would like to single out Tony Mullen and Rob Malouf, for all the stimulating discussions we have had over coffee or beer. And for being very good office-mates; although on that particular front I have to say that I have nothing but good memories from my current office-mates as well, Begoña Villada Moirón and Susanne Schoof.

I would also like to thankfully mention Ashwin Srinivasan for writing Aleph. And all the people outside the department who have shown interest in my project and with whom I have had fruitful discussions and especially Rui Camacho, Vitor Santos and all the people in Porto. I would also like to thank Pavel Brazdil, the director of the AI lab in Porto, for offering me hospitality at the lab.

On the non-computational side of my project, lots of thanks to Wouter Jansen, Dicky Gilbers, and Roberto Bologniesi for every discussion we have had about Phonology and every bit of ignorance and lack of linguistic background they have helped dissolve. And to the Zeppelin for helping me clarify some Logic Programming concepts in my head last March. And to Kengo Harimoto for his explanations and bibliographical references to Indian Logic.

It would also like to thank the members of my reading commitee, Walter Daelemans, Theo Kuipers, and Gerard Renardel de Lavalette; their comments and suggestions have had a great impact at improving the final version

of this thesis.

My gratitude also goes to Rob Visser, Anna Hausdorf, Wyke van der Meer and the *Secretariaat CNL* for their administrative support. And Shoji Yoshikawa in *Letteren* and Kees Visser in HPC for their systems administration: their contribution to creating a smooth working environment is very much appreciated.

There is also lots of people whom I am indebted to, although they were not directly related to my project or to this thesis. First of all I should thank Willem Moolenburgh for my being here in the first place: his enthousiastic description of life in Groningen played a great role in overcoming my original reservations about making such a long-term commitment to a small town. And then, of course, all the people who made Groningen live up to Willem's enthousiasm starting from Tony and Ivo who were the first people I met here; all the people who have joined me for a cup of coffee and a smoke at the last table to the right, on the sunny side of the smoking section of the Harmonie Kantine; my flatmates from 4Zuid to Kl. Raamstraat to Oosterparkwijk; Алёна, Kanat, Laura, Марина, Simo, the Wrocław students of Dutch, the ex-YUs, two generations of EMCL and Euro-Culture students, and all the people I've met in Groningen. And, of course, a special mention should go to *Vera*, *Filmcentrum Poelenstraat*, and the *Paard van Troje:* Groningen just simply wouldn't have been the same without these fine establishments.

A special thanks goes to Leonoor van der Beek and Eleonora Rossi, my paranimfs, for their help with organizing everything so nicely. Well, strictly speaking for their agreeing to organize everything so nicely in the near future, but I'm sure they'll do a great job. Thanks once again to Leonoor for by far the best *samenvatting* translation I've come across in my 5 years here, but also for her comments that greatly improved the English original as well.

And, of course, a big thanks to Θεμιστοκλή and Ελευθερία, my parents, for their support, and for my being here in the first place!

Finally, I would like to close with a somewhat unrelated comment: if there is one single thing the whole Groningen experience tought me, it's that the Ukrainian redberry is the sweetest and juiciest fruit there is.

That was it. If I forgot somebody, my most profound and sincere apologies, I didn't mean to!

I hope you'll enjoy reading my thesis.

Groningen, 11-10-2003

Contents

Chapter 1

Introduction

Induction in Logic and Philosophy is the process of establishing natural laws by reasoning from the parts to the whole, from the particular to the general, or from the individual to the universal. Inductive reasoning collapses the aggregate of all observations or instances to the much shorter law that describes them and is able to *deductively* predict them. *Deduction* is, then, reasoning in the opposite direction from the natural laws to the observations, from the whole to the parts, from the general to the particular from the universal to the individual.

The *Oxford Dictionary of Philosophy* [3] defines induction as:

> any process of reasoning that takes us from empirical premises to empirical conclusions supported by the premises, but not deductively entailed by them.

In other words, induction forms a *hypothesis* that explains a set of empirical data or observations. The observations *support* the hypothesis (that is, do not contradict it) but do not *prove* it; induction is an *ampliative argument* that enlarges the scope of its premises and thus generates knowledge, in the sense that it derives a general law from specific data.

Induction lies at the core of *Epistemology* — the theory of knowledge, the investigation of the cognitive process of knowledge discovery — which in its turn has always been an integral part of Philosophy and Logic. Induction is also the object of Machine Learning, the field of Computer Science which is attempting to automate knowledge discovery. *Inductive Logic Programming* is such an attempt to automate the induction of empirical rules, where the description formalism is Predicate Logic.

At the same time, understanding and codifying the rules that govern language has also been at the focus of philosophy, cognitive science and — of course — linguistics since their very early days. Since the seminal work

of Chomsky [12] in the 1950s, Predicate Logic (and especially its *Unification Grammar* incarnation) has been used to describe various aspects of language, and especially syntax.

It is only natural, then, that the combination of Inductive Logic Programming and Linguistics is a very tempting direction of investigation. This work is an account of the author's submission to this very temptation, and a description of its fruits.

1.1 Overview

This book is divided into six chapters, including the *Introduction.* The remainder of this chapter is a historical overview of induction in epistemology and pre-formal logic, where background ideas and concepts that will be important in the discussion of ILP in Chapter 2 are introduced.

The second chapter, *Inductive Logic Programming,* introduces *Inductive Logic Programming* (ILP), the discipline of Machine Learning that deals with the induction of First-Order Predicate Logic Programmes. The simpler tasks of learning in Propositional Logic formalisms (namely, Decision Trees and Rule Systems) are explained first, introducing ideas and concepts that culminate into ILP once extended in the domain of Predicate Logic. ILP is discussed at the theoretical level, as well as by providing examples of concrete algorithms and implementations. Particular weight is given to one implementation, ALEPH, which is the ILP system used throughout the work described here.

One of the largest problems with ILP is its requirements in computational power and memory capacity, which can be alleviated by parallelising the task. The third chapter, *Data-Parallel ILP*, describes a data-parallel variation of the algorithm developed in the course of this work. Its advantages, disadvantages and scope of applicability are discussed. The work described in this chapter was also presented in a parallel and distributed Machine Learning workshop [37].

The following two chapters deal with the application of ILP to two linguistic tasks: *Shallow Parsing* (Chapter 4) and *Phonotactics* (Chapter 5); the tasks are explained and the experimental setup described, and the results of applying ILP to these tasks are presented and discussed. The motivation behind the choice of these two tasks is that they are both treating local phenomena. That is to say, phenomena that can be explained with theories that do not rely on either long-distance dependencies or deep structures. The work on shallow parsing was originally published in the Proceedings of the Computational Linguistics in the Netherlands conference [33, 36]. The work on

Phonotactics was originally presented in the Student Session of ESSLLI 2001 [34] and subsequently published as special issue of WEB-SLS [35].

The final chapter summarises and discusses the conclusions drawn from each chapter individually, but it also draws general conclusions from this work as a whole.

1.2 Deduction and Induction

Let us consider the classic example of deduction where from the clauses:

1. All humans die.

2. Socrates is human.

the fact that Socrates dies can be deduced. In this sense the first clause is a theory of mortality, a rule for predicting which individuals will die. One other way of looking at deduction is as an explanation of Socrates' death. This notion of deduction as explanation stems from the fact that statements like *all humans die* are seen as implying a cause-and-effect relation. These two perspectives of looking at a clause agree in that the best way to predict the behaviour of a system, is to know its internal workings and to be able to explain why each phenomenon happens. The second clause, on the other hand, can be seen as a fact, a piece of information about the world, which is used to satisfy the premises of the mortality theory.

Inductive reasoning would, then, reverse the direction of deductive reasoning by using world knowledge and a set of observations to construct a hypothesis from which one can deduce (explain) the observations given the world knowledge. Scientific discovery is such an inductive process which tries to formulate theories which explain the observations (our experience) given our already existing knowledge.

The mechanics of this inductive process have been the subject of philosophic and epistemological discussion for centuries — and still are. Very roughly speaking, the *rationalist* schools of thought argue that we should accept as knowledge only what is deductible from axioms that are in some way innate and inescapable, whereas more *empiricist* schools will argue that scientific knowledge consists of laws that correctly and consistently predict our experiences, and their validity does not need any further ideological support. This section will very briefly summarise the course of the arguments of rationalism and empiricism, but with a noticeable bias towards the latter, since it forms the historical and philosophical background of ILP. The reader

is referred to the historical background chapters of Stewart Shapiro's *Philosophy of Mathematics* [75] and to Encyclopedias of Philosophy [3, 89] for a more detailed and balanced discussion.

1.2.1 Aristotelian Logic

One of the earliest — and through the centuries by far the most influential — attempts to formalise argumentation and proof is Aristotelian logic. Aristotle's *syllogismos* almost perfectly matches our modern notion of deduction: in his *Prior Analytics* (I.2, 24b, 19-21) he defines *syllogism* as:

> λόγος ἐν ᾧ τεθέντων τινῶν ἕτερον τι τῶν κειμένων ἐξ ἀνάγκης συμβαίνει τῷ ταῦτα εἶναι.
>
> a form of words in which, when certain assumptions are made, something other than what has been assumed necessarily follows from the fact that the assumptions are such.

as translated by Tredennick [85]. In modern notation, a syllogism is a statement in the form of '**if** P_1, P_2 ... **then** C' where the truth of the *premises* P_1, P_2 ... necessitates the truth of the *conclusion* C.

Aristotle proceeds to identify *epagoge* 'induction' as the other form of argumentation besides deduction. He simply characterises it as 'argument from the particular to the universal' in *Prior Analytics*, but induction plays a central role in his *Posterior Analytics* [86] where it is recognized as the basis of the theory of knowledge.

Aristotle argues that we accept propositions as true when we can deduce them from true premises that exhibit cause-and-effect relationship between the premises and the conclusion. This, naturally, only pushes the problem of scientific knowledge back to the knowledge of these more elementary prior propositions, the axioms, which themselves require justification, creating the problem of infinite regression. In other words, if scientific theories are to be proven by deduction, they have to be proven from some more fundamental and general propositions, which in their turn also need to be deduced from even more fundamental and general propositions, and so on ad infimum. The *agnostics* solved the problem by allowing circular proofs, in which it is possible for premises to also be conclusions. Aristotle opposed this view and instead advances the notion of axioms from which all scientific theories can be proven, without attempting to offer any means of justifying our choice of axioms.

Aristotle in these writings is effectively establishing the foundations of empiricism, by justifying our choice of axioms (and, consequently, scientific

theories) through their ability to support what we know and observe to be true propositions. In this he is coming in direct contrast with Plato's rationalism, which refers to *ideals* and innate knowledge.

1.2.2 Rationalism and Empiricism

The argument regarding the origins of the axioms of logic and scientific systems was heightened during the 17th century. On the one hand, Rationalists such as René Descartes and Benedict Spinoza argued in favour of innate concepts of causality and mathematics from which all scientific knowledge deductively derives, whereas Empiricists (John Locke, David Hume, and Thomas Reid[1] among others.) claimed that all knowledge, no matter how abstract, ultimately traces back to experiences.

To derive abstract concepts and knowledge from concrete observations, empiricism refers to Francis Bacon's *true and perfect induction*, described in his 1620 *New Organon*. Bacon suggests the synthesis of abstract knowledge by careful generalizations of observations; each generalization step is checked by deductively deriving from it all propositions that must be true if the generalization is correct and comparing them with our experience and observations.

Immanuel Kant tried to reconcile the two movements in the 18th century, by identifying two kinds of knowledge: *analytic* knowledge is tautological (for example, *All mortals die*) or constructed from previously known propositions by deductive inference tools that are guaranteed to draw correct results from correct premises. This web of analytic proofs leads back to a set of core propositions — the axioms of the system — that are taken to be true without proof within the system.

Axioms are either carried over from a different system or are known *synthetically*. Synthetic knowledge is non-tautological propositions like *All humans are mortal*, which are not known deductively and are not rigorously proven to be correct, but are constructed *(synthesised)* empirically and when they are based on enough observations they are stable enough to be correct beyond reasonable doubt.

Kant also argues for knowledge that is 'synthetic a priori' and 'intuitively known', especially in the fields of mathematics and geometry, but that is only tangential to our discussion of induction.

[1] It was Thomas Reid, in fact, who was the first one to articulate the rationalism versus empiricism divide in his *Inquiry Concerning the Human Mind*.

1.3 Inductive Inference Operators

The empirical line of investigation presented above focuses on the importance of experience in the formulation and verification of natural laws, but does not delve into the issue of the mechanics of induction; that is, the logical tools and methods that should be applied to form and support a hypothesis.

As McCosh [42, p. 8] points out, early empiricists such as Locke do not make explicit reference to induction as a method of scientific discovery, and this development only came about later through the work of Thomas Reid and the *Scottish School of Common Sense* in the 18th century.

It is, however, in the 19th century and J. S. Mill's *System of Logic*[2] that inductive operators are systematically presented and analysed [45]. Mill's inductive operators aim at modelling the way that scientific research is carried out and true propositions regarding nature are discovered.

Mill builds a system of Logic that systematises scientific argumentation and methodology. He starts by distinguishing specific-to-general inductive inference from general-to-specific deductive[3] inference as well as from perfect, mathematical induction (Chapter 2: *Of Inductions improperly so called.* See also Section 1.5 below.) and defends induction and empirical scientific methodology against rationalism, as represented by Whewell [88] during his times.

Mill calls atomic observations *events*, and one such event will be the *phenomenon* that is being investigated. A complex observation involving many co-occurring events will be an *instance* of the phenomenon, if the phenomenon is one of the events of the observation. So, for example, given events e_1, e_2 and e_3, if e_2 is the phenomenon we're interested in then observing e_1 and e_2 co-occure is an instance of e_2, whereas observing e_1 and e_3 is not. Although it is clear that the goal is to establish the causes of the phenomenon, this more neutral formulation is chosen so as to avoid implicitly making any assumptions on cause-and-effect relations between the events being discussed or the direction of such relations.

With this remark Mill is clearly anticipating the concept of *features* in the Machine Learning literature. (See also Section 2.1, and particularly 2.1.4.)

But what is of particular interest to our discussion is his inductive operators[4] each one formalising[5] a scientific *method* of deriving physical or chemical

[2] Secondary sources for Mill's work include textbooks of Logic written by his contemporaries after his work — for example by Jevons [31] — as well as more modern work like that of Skorupski [78].

[3] *Ratiocinative* in Mill.

[4] *Canons* in Mill.

[5] 'Formalizing' from Mill's epistemological perspective, since the Logic formalization will only come after the work of Boole and De Morgan has introduced Formal Logic. Mill

laws from observations that has been well-tried and firmly established practice in the scientific community.

The first inductive operator corresponds to the *method of agreement*, that is, the scientific practice of looking for cause-and-effect relations between consistently co-occurring events. More specifically, the operator is defined as:

Definition 1 First Operator: *if multiple instances of a phenomenon have only one event in common, that event is either the cause or the effect of the phenomenon.*

The *method of difference* is based on the reasoning that if an event is always present in a phenomenon's instances, and the phenomenon is never observed in its absence even when the remaining conditions are kept constant, that event must have a causal relation to the phenomenon. The operator describing this method of investigation is stated thus:

Definition 2 Second Operator: *if a set of events co-occurs with a phenomenon, but the same set save one event does not, then that event is either the effect or part of the cause of the phenomenon.*

These two operators are an elaboration of the Baconian method of *elimination* of events that cannot have a cause-and-effect relation to the phenomenon under investigation, when they can be absent in the presence of the phenomenon. The first operator is formulated so that it can be suited to observation of naturally occurring phenomena, whereas the second is more focused on controlled experiments. These two operators are also combined into the third *joint method of agreement and difference*.

As an attempt to relax the strong requirements of the method of difference, the *indirect method of difference* is effectively the method of agreement with the corroborative evidence of observations where the absence of the agreeing event co-occurs with the absence of the phenomenon, the event is either the effect or part of the cause of the phenomenon. The corresponding *Third Operator* will not be dealt with any further here.

The *method of residues* is a corollary of the method of difference, where independently established laws are employed to deduce the negative instance (where the phenomenon is not present.) An alternative way of looking at this method is that it is isolating the unexplained residue after all the prior knowledge available has been applied:

---

was, in fact, aware of De Morgan's work but dismissed Formal Logic as simply a subset of Logic as a whole [44, Chapter 3, par. 9].

Definition 3 Fourth Operator: *from the observation that a set of events co-occurs with a (complex) phenomenon, remove all the atomic events of the phenomenon that are known to be the effect of some of the antecedent events. The remaining antecedents have a causal relation to the residue of the phenomenon.*

Finally, the *method of concomitant variations* is a quantitative variation of the method of the method of difference: instead of the observation that the presence of an event is linked with a phenomenon, it is based on the observation that a quantitative change in an event coincides with a quantitative change in the phenomenon:

Definition 4 Fifth Operator: *if variations of any quantitative aspect of a phenomenon coincide with variation of any quantitative aspect of an event, the phenomenon and the event have a causal relation.*

1.4 Induction and Justification

Since empirical theories are not deduced by sound derivations, but are, formally speaking, left unproven, the problem of justification for them arose quite early. The obvious requirement is that the hypothesised theory is sufficient to deduce the observations: as we already saw, Aristotle already put forward the notion that scientific endeavour is the search for an axiomatic system that can deductively explain our experiences and Bacon noted the need for extensive deductive confirmation of empirical theories by the data.

Mill, in particular, notes that the deductive verification against the observation is essential, and that

> for this reason, the method of science has often been called a deductive-inductive method. Indeed, pure induction is probably inconceivable, since we cannot enter upon a mental process unless we first entertain some general ideas. Induction and deduction are interdependent functions of the ratiocinative mind.

Finally, Popper [62] puts forward the idea that an empirical theory must be *refutable,* but nevertheless not refuted. Furthermore, the more restrictions are deducible from the theory, the more easily refutable it is since it offers more opportunities for counter-examples to be discovered, and the safer we are that it is correct in the absence of such counter-examples.

This evaluation of a theory based on performance (accuracy of prediction) is, however, not enough to capture a theory's generalization performance. For example, consider the theory that *all humans die* that solves the problem of

explaining Socrates' mortality given that he is human. This solution is not the only accurate one as far as we can tell from the data, if we assume, for example, that our observations include Socrates and Plato as examples and Zeus as a counter-example of the phenomenon we are trying to induce. Then we see that multiple solutions (e.g. *All philosophers die*, or *Only Socrates and Plato die*) fit the data and the prior knowledge.

This introduces the problem of justifying a particular choice of a theory among the ones that fit the data, and the elegance or simplicity of the solution has also been considered an indication of correctness since the times of Aristarchos who, already in the 3rd century BCE, favoured the heliocentric model over the geocentric one 'for aesthetic reasons'.

But the person who has mostly linked his name with this methodological principle is the 14th century philosopher William of Ockham (also spelled Occam) after whose razor the principle of *Ockham's Razor* was named. Although Ockham himself was only interested in not adding superfluous elements in an ontology (see [48] for more about Ockham), Ockham's Razor is the very general scientific methodological principle of favouring simpler theories.

But it is obvious that this *principle of parsimony*, as Ockham's Razor is also called, is only a methodological guideline. First of all because it is based on the elusive and somewhat subjective notion of simplicity. Furthermore, even if consensus is achieved on an objective measure of simplicity for a particular problem, this will often mean being able to break a problem down to its elementary blocks or otherwise require a thorough prior understanding of the problem and its possible solutions. We will see in the discussion about the *evaluation functions* (Sections 2.2.3 and 2.3.4 below) how the issue of justification of a choice among all the possible theories that fit the data is handled in ILP.

1.5 Perfect Induction

One point that is being repeatedly made in epistemological discussions is the distinction between induction and *perfect* or *mathematical induction.* Perfect induction is composing rules by *deductively* verifying them against *all* possible instances, and thus is not ampliative. That is, it does not extend the scope of our knowledge, it is a theorem rather than a new piece of empirical knowledge.

This distinction is explicitly made by both Kant and Mill. Kant argues that only synthetic knowledge is new knowledge, (deductive) analysis only

offers different ways to look at the same body of known propositions. Mill[6] accordingly adds in Book 3, Ch. 2, *Of Inductions improperly so called* that perfect induction is reasoning from *all* of the parts to the whole, and is thus not synthesising any new knowledge. For that reason, Mill suggests that only imperfect induction will be called induction and will be dealt with in his chapter on inductive operators.

One observation that needs to be made here is the distinction between the quantitative and the qualitative evaluation of a theory. The arguments above are quantitative in that they are based on the coverage of the induced theory, and consider knowledge as new only if it can expand this coverage, or scope, of the theory. This view is also consistent with Information Theory and its definition of information content through entropy: the more 'surprising' or 'unlikely' a message (or proposition) is, the more information it carries; therefore, propositions that are already deducible by the system have no information content. (See Section 2.1.2 for more on entropy, although in a different context.)

One should note, however, that the qualitative properties of a theory are also interesting, and inferring a short rule from a long or infinite collection of data does add to our knowledge; or, more accurately, improves the quality of our knowledge. In the case of mathematical induction, an infinitely long series of propositions is collapsed into an equivalent closed formula. Another example is knowledge migration or inducing rules from an oracle: even if a perfect model of a system exists, and the system's behaviour can be correctly and accurately predicted, it might be interesting to use examples retrieved from this model to induce a new one in a different formalism that might be for some reason more interesting or more easily applicable.

[6] A distinction that, amusingly enough, Jevons fails to explain in his textbook [31] based on Mill's *System of Logic*. In Lesson 26, p. 227 we read: 'Mr. Mill considers Geometrical and Mathematical Induction not to be properly called Induction, for reasons of which the force altogether escapes my apprehension; but the reader will find his opinions in the 2nd chapter of the 3rd book of his *System of Logic*.'

Chapter 2

Inductive Logic Programming

In the introductory chapter we saw a series of pre-formal attempts to analyse and understand induction and the knowledge-construction process in general. The use of computers to perform symbolic calculation has, however, opened a new chapter in understanding knowledge acquisition and — at the more practical level — the automatic synthesis of knowledge.

This chapter starts by describing the most important approaches to inducing propositional classifiers like Decision Trees (Section 2.1) and Propositional Rule Systems (Section 2.2). This introduction eases the way into the section describing Inductive Logic Programming (ILP) in general (Section 2.3) and the single-predicate learning ILP algorithm PROGOL in particular (Section 2.4). The chapter closes with a quick reference to other kinds of ILP (Section 2.5) and some thoughts on the domain of applicability of ILP, which will be further elaborated in subsequent chapters describing the application of ILP to particular tasks.

2.1 Decision Tree Induction

Research on Propositional Logic induction and ILP has been (and still is) greatly influenced by work done on decision trees. Furthermore, certain concepts that show up in the more complex environment of propositional or first-order induction will be more smoothly introduced and more easily understood in a description of decision trees and decision tree induction algorithms.

Decisions trees are *classifiers* that are presented with objects which they must classify as belonging to one of a predetermined set of classes. The basis of the decision is the values of one or more characteristics or *attributes* of the objects. These attributes must be sufficient to fully specify each individual

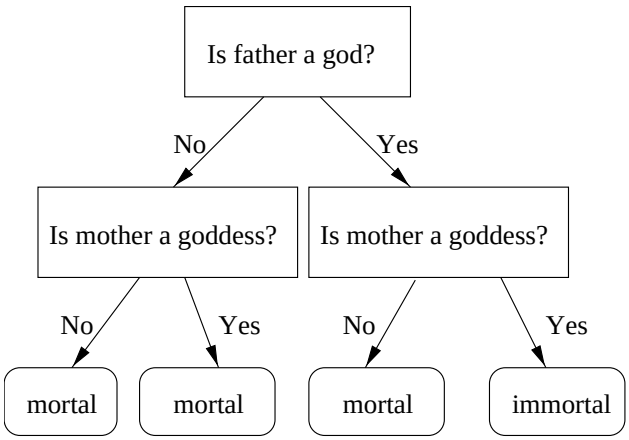


Figure 2.1: A Simple Decision Tree

object. Each node in a decision tree carries the name of an attribute and branches off to a different child node for each possible value of that attribute. The classification proceeds from the root node towards a leaf by examining each node to see which attribute it refers to and then comparing the object's value for that attribute with the values on the branches leading off from the node. The one that matches is followed and the process is repeated for the attribute of the new node, until a leaf is reached. All objects arriving at a particular leaf should then be of the same class, and that class will be the decision tree's prediction of what the class of the object is.

The simple decision tree of figure 2.1, for example, classifies individuals as mortal or immortal according to their ancestry. This decision tree can be seen as defining the *concept* of being mortal or immortal, in the sense that a concept is a rule that allows individuals to be classified as instances of the concept or not.

The task of machine learning is to process a set of observations, the *data*, to discover the definition of the *target concept* that these observations are instances of. One way of looking at it is that data is being collected by observing a process the mechanics of which we do not understand. This data forms the extensional definition of the target concept. We are interested in using this extensional definition as evidence to hypothesise about the intensional definition of the same concept, in the hope that it will help shed some light in the mechanics of the underlying process. In our example a machine learning algorithm would formulate a hypothesis about the mechanics of death based on a list of individuals who have died and a list of individuals

who haven't.

The target concept is the correct solution to our problem, since it is, by definition, accurately modelling the process that is generating the data. It is obviously impossible to prove whether the hypothesis constructed by any machine learning algorithm — or, for that matter, human — is identical to the target concept or not, but it is possible to estimate the extensional proximity of the two by using the data to evaluate the theory. In other words, it is possible to use the hypothesis to classify data the real classification of which we already know, and then compare this classification with the actual observations.

The earliest approaches at automated knowledge induction by Hunt et al. [29] in the 1960s were directed at decision tree induction and were inspired by psychological experiments conducted ten years earlier by Bruner et al. [9] rather than the epistemological discussion on induction and knowledge synthesis. To formalise the task, Hunt et al. [29, Chapter 1] define the following terms, based on previous definitions by Church [13]:

Definition 5 Objects *are the entities which may be observed by a concept learner. The set of all objects is the* universe.

Definition 6 *Objects can be distinguished, one from the other, by the value they have on certain* attributes. *Thus an attribute is a dimension of variation of an object.*

Definition 7 *The* description *of an object is a statement of its position on all observable attributes (dimensions).*

Definition 8 *A* value *is a set of those positions of an attribute which are to be treated equivalently.*

Definition 9 *A* description space *is the set of all possible descriptions of objects. Since an object can have at most one position on a given attribute (and, hence, have only one value) the description space is the set of statements formed by combining one value from each attribute.*

Definition 10 *A* name *is an arbitrary label attached to every object in a particular subset of the universe. A* denotation *of a name is the set of objects to which the name can be applied. A* concept *is a decision rule which, when applied to the description of an object, specifies whether or not a name can be applied.*

Definition 11 *A* concept learning system *(CLS) is a device for creating a concept corresponding to some partition of a sample of objects which have been categorised by a pre-established rule for using a name.*

The 'sample of objects' of this last definition will be called the *training data.* The 'rule' that generates this training data is not necessarily an explicit one, but it could also be any experiment or process. In the former case of an explicit rule, the CLS is performing *knowledge migration* from one representation to another, in the latter *knowledge discovery* from empirical data.

This formulation of the task is trying to be as generic as possible within the framework of symbolic machine learning, by referring to a generic CLS that is constructing some 'decision rule' (concept). It, however, clearly mirrors decision tree induction, by defining the object representation formalism as tuples of attribute values and the decision rule as a rule operating on such tuples.

2.1.1 The Original Concept-Learning Systems

This class of decision tree induction algorithms is based on the assumption that the instances of each class of objects will tend to have more common features (that is, have the same value for more features) than instances of different classes. The learning algorithm should, then, search the description space for sub-spaces where instances of each concept tend to cluster closely together.

The general strategy used is a divide-and-conquer approach that constructs a tree from the root towards the leaves, as shown in Algorithm 2.1. The most important aspect of these algorithms that is left under-specified in the general schema above, is the choice of the 'best attribute' for each node. The various metrics proposed are trying to pick the attribute that will lead to homogeneous leaves with as small (shallow) a tree as possible, basing the decision on the distribution of instances of each class among the active examples.

The Concept Learning System (CLS) series of algorithms [29, Chapters 3–5] were the first such learners. CLS-1 through CLS-3 learn binary trees, where the objects of the universe are only classified as positive and negative instances of the concept. The algorithms would go through the positive objects, counting the frequency with which attributes $\times$ values occur in their description. The most frequent one is chosen as the test for the root node of the decision tree, and the process is recursively repeated for each of the two sections in which the training data is partitioned by this test. The algorithm terminates when all branches have reached homogeneously positive or homogeneously negative leaves.

CLS-4 and CLS-5 were relying on an *oracle,* rather than training examples: the learning algorithm remains the same, but the partial trees are evaluated

Algorithm 2.1: General Strategy for Decision Tree Induction

if all active examples are of the same class **then**
  Make the current node a leaf and assign this class to it.
else
  Pick the best $A \in$ *Attributes*.
  for all values a_i of A **do**
    Start a sub-tree from the current node.
  end for
  Split the remaining examples among the resulting
  trees, depending on their value for A.
  Recursively apply this algorithm for all such subtrees.
end if

on a data set chosen by the learner. The data set construction algorithm asks the oracle to classify random objects, until one is found where the partial tree makes an error. The tree is retrained so that it classifies this new object correctly, and then objects in the neighbourhood[1] of the misclassified objects are tested and the tree re-adjusted after each misclassification.

CLS-5, in particular, *refines* the tree hypothesis, rather than retrain a new tree: errors are due to a leaf not being homogeneous, and so the tree is extended under the leaf which caused the misclassification. This is, in effect, implementing a general-to-specific hill-climbing search through the tree hypothesis space.

CLS-6 experimented with more informed heuristics than attribute-value frequencies, taking the negative instances into account as well: if P is the number of positive instances having this attribute-value in their description and N the number of negative ones, then *coverage* over the remaining training data is defined as $P - N$ and *relative frequency* as P/N.

Coverage is found to perform much better, because relative frequency tends to be unstable when the data gets sparse near the leaves. More complex statistical methods are also suggested but not further explored: the χ^2 criterion and, most importantly, the *transmitted information* when the class membership predicted by the decision tree is viewed as a signal, given the description (see 2.1.2 below for more). CLS-7 and 8 were variations of CLS-6 where the notions of 'positive' and 'negative' might be reversed if it allowed for better, more compact theories.

CLS-9 was, finally, an extension that could induce not only binary trees,

[1] With distance being defined as the number of attributes where the two descriptions differ.

but trees of arbitrary branching. This was accomplished by generalizing the heuristics that choose which attribute to branch on next: the training examples that are taken into account are the ones that are still active on the current node, i.e. the ones with a description matching the partial description specification from the root to the current node. Each attribute at a node would create a partition of the examples active at the node, such that each subset of the partition is homogeneous with respect to the value of the attribute. For all remaining attributes, the homogeneity with respect to the class of the examples is estimated as

$$H_i = \sum_j \max_k n_{ijk}$$

where n_{ijk} is the number of instances that are still active on the current node, have value j for attribute i, and are in class k. The attribute i with the highest H_i is then chosen.

2.1.2 Entropy as Search Heuristic

Based on CLS, and exploring further the advantages that more informed heuristics would have to offer, the ID3 algorithm [64, 65] borrows the concepts of information content and entropy from information theory to estimate the homogeneity of a partitioning. This is done by viewing the class under which the tree classifies an object as a signal emitted by the tree, and calculating the amount of information carried in the signal.

Entropy is a measure of the lack of order, originally in thermodynamic systems. Shannon [74] introduced the concept in information theory[2] and defined it as the expected (on average) number of bits required to encode a message, using the most efficient encoding possible. For an alphabet of k distinct symbols, appearing with relative frequencies p_i, $i = 1..k$, the *information content* of symbol i is $-\log_2 p_i$ bits. The entropy of the whole encoding is then the weighted average of the information contents of all symbols:

$$H = -\sum_{i=1}^{k} p_i \log_2 p_i$$

and estimates the number of digits necessary to transmit each symbol. Low values of entropy imply a high level of organization, the existence of patterns in the signal, and high compressibility.

[2] Cover and Thomas [20] have more on entropy in information theory.

When the concept of entropy is transferred to a prediction made at a decision tree's node, then the 'signal' is the class membership of the object under classification. The prior probability of the object's belonging to a class is the fraction of the active training data that is in that class. The choice of attribute A for a particular node should, then, be based on the decrease of entropy achieved by partitioning the active data according to their values for A.

If $H(D)$ is the entropy of the active training data D, D_i is the subset of D with all the objects with value a_i for feature A in their description, then the *information gain* for branching on A at this node is:

$$Gain(D, A) = H(D) - \sum_i \frac{|D_i|}{|D|} H(D_i)$$

This information gain estimates how informative attribute A is at this node of the decision tree, or, in other words, how much it contributes to getting closer to being able to make a decision (i.e., reach a homogeneous leaf.)

The $H(D)$ term is constant for all features, so the feature that minimises the summation is the one that achieves the highest information gain at this node. The ID3 algorithm then proceeds in the same way as CLS, by building a decision tree from the root towards leaves where only examples belonging to the same class are active.

The qualitative difference that ID3's heuristics make is that now the function estimating the homogeneity or 'purity' of a partitioning is taking into account all the members of each partitioning, whereas in the CLS series only those belonging to the most populous class were counted.

2.1.3 Avoiding Overfitting

The CLS algorithms were designed to learn the training data perfectly, in other words the tree-building process would only terminate when all the training examples were correctly classified. It was observed, however, that this approach is not well-suited to real-world data, since they tend to be *noisy*, i.e. including errors and inconsistencies.

But even with correct and consistent data, fitting the training data perfectly often results in highly specialized — or even fully specified — paths that fit only a handful — or even just one — example. Such highly-specific paths are very unlikely to be taken when the tree is used on unseen data, meaning that the tree will perform very poorly on unseen data.

To avoid this problem of *overfitting*, inductive algorithms need to decide when the accuracy on the training data is 'good enough' and further training

will only make the overall performance worse. In later versions of ID3 this is done by setting a threshold of information gain, below which adding a new attribute is not considered interesting any more. Generally speaking, modern decision-tree learning algorithms will stop extending the tree as soon as the gains of further partitioning become too low.

2.1.4 Some Thoughts...

One thing that should be noted is that the CLS algorithms partially rediscover[3] J. S. Mill's results. The first three CLS algorithms in particular, closely resemble Mill's *method of agreement* (Definition 1, Section 1.3) in that they both attribute cause-and-effect relationship between consistently co-occurring events.

It is interesting to note that Mill as well as the designers of CLS are not trying to invert deduction, but rather to logically formalize what human researchers are (or should be) doing when suggesting a scientific theory. In the ILP tradition, on the other hand, similar results are reached by building upon a formalization of deductive reasoning and, by inverting it, deriving from it a formalization of inductive reasoning without making any psychological or epistemological references.

2.2 Propositional Rule Induction

Decision Lists or *Rule Sets* or *Propositional Rule Systems* (for the rest of this chapter, simply *Rule Systems*) are lists of if-then rules, that classify examples based on the values of their attributes.

The rules are of the form:

IF $A_i = v$, $A_j \neq u$ **THEN** *class*

where the antecedent, or *body*, is a conjunction of attribute tests and the consequent, or *head*, the class or concept membership of the object being tested.

The elementary *literals* that rule bodies are made of can test either equality or inequality between the value of an object's attribute and a constant, operating in identical fashion as decision tree nodes.

Multiple rules (or *clauses*) can be specified for each class, which are interpreted disjunctively and express alternative 'sorts' or 'kinds' of the object's

[3] *Re*-discover because there is no reference to Mill or formal logic or previous inductive theories, only psychological experiments conducted by the authors and Bruner et al. [9].

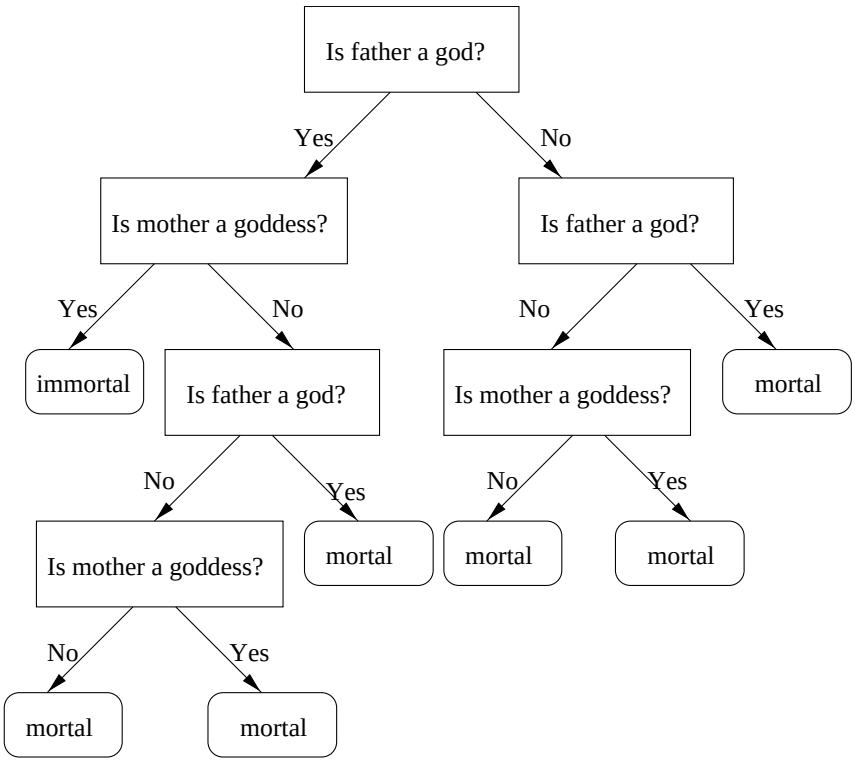


Figure 2.2: A redundant Decision Tree representation of the deity-ness classification Propositional Rule System.

instances. In their simplest form[4] the rules are ordered (that is, they are 'if-then-else' or 'first-match-wins' rules). When presented with an object to classify, the rule system will go through the rule set until the first matching rule is found. The class in that rule's consequent is then returned as the object's classification.

The descriptive power of rule systems is equivalent to that of decision trees: a tree can be easily transformed into a rule system where each path from the root to a leaf is spelled out as one rule with the class of the leaf as its consequent. The decision tree in Figure 2.1 for example would be written

[4]Some systems have weights attached to the class, representing the level of confidence of the classification suggested by each clause. Objects are tried against *all* rules, and the class weights offered by each matching rule summed up. The object is then classified as belonging to the class with the highest sum. These rule express different aspects or characteristics of a concept, rather than different sorts or sub-classes of its instances. This 'stochastic' kind of rule systems is not dealt with here.

as the following equivalent rule system:

```
IF father_is_god == no  AND mother_is_goddess == no
  THEN mortal
IF father_is_god == no  AND mother_is_goddess == yes
  THEN mortal
IF father_is_god == yes AND mother_is_goddess == no
  THEN mortal
IF father_is_god == yes AND mother_is_goddess == yes
  THEN immortal
```

In the other direction, rule systems can also be represented by binary decision trees as follows: each conjunctive rule is a path to a leaf. Each positive outcome for the test in a node leads to the next conjunctive test, and each negative outcome leads to the path of the next (disjunctive) rule. For example the rule system above can be collapsed to the following three rules:

```
IF father_is_god == yes AND mother_is_goddess == yes
  THEN immortal
IF father_is_god == no  AND mother_is_goddess == no
  THEN mortal
IF true THEN mortal
```

which will be expressed as the tree in Figure 2.2. Note that this decision tree defines the same concepts as the one in Figure 2.1, and it will also become the same tree once all the redundant checks are removed.

Despite their equivalence in expressive power, decision trees and rule systems differ conceptually, in that they represent different[5] ways of approaching the decision-making process. Decision trees separate instances into increasingly smaller and more homogeneous groups, and are seen as a whole rather than as a collection of paths from the root to the leaves. Rule systems, on the other hand, are collections of rules and each individual rule captures a different class. In this way the classification of each instance is determined (and, in a sense, justified or explained) by one rule — the one that matches the attribute values of the instance — rather than the classification system as a whole. Since the ordering of the rules is important for the classification of instances matching the antecedents of multiple rules, some interaction between the rules is unavoidable, but rule systems are still more 'modular' and provide a more explicit 'definition' for each concept.

[5] In an analogous way, the Java Virtual Machine and the Warren Abstract Machine, for example, are both Turing machines.

One more thing that should be noted regarding the descriptive power of rule systems is their relation to Propositional Logic. One can think of all the rules that yield the same class as a disjunction, since they represent alternative ways for an instance to be a member of that class. The literals in the body of each rule are logically conjuncted, since they all have to be true for the instance to match the rule. In this way, a rule system can be seen as a set of Disjunctive Normal Form definitions, which means that they implement full Propositional Logic. Note that Rule system implementations have varying behaviour when multiple rules match an instance, so there will be varying degrees of divergence from the expected behaviour of a Propositional Logic programme.

2.2.1 Learning Lists of Rules

Given the conceptual difference between Rule Systems and Decision Trees, rule-learning algorithms are more oriented towards the construction of individual rules than the refinement of the classifier as a whole, despite the fact that rule-induction systems borrow heavily from ideas developed for decision-tree induction. Rule-by-rule learning is, for example, already hinted at by CLS-5, where only individual paths (that is, rules) were refined after each mis-classification, rather than retraining the whole tree.

Rule-induction machine learning systems employ two algorithms: a high-level algorithm for building up the system from individual rules, and a low-level one for constructing each rule. The high-level algorithm is based on the generic *sequential cover* strategy, whereas the low-level algorithm is one of the standard partially-ordered search algorithms.

Sequential cover is a greedy strategy for incrementally building a rule system one rule at a time. Rules get constructed by the low-level algorithm, using only the active (that is, still remaining uncovered) positive examples. As more rules get appended to the theory, the system's coverage expands and examples get removed from the still-uncovered example pool. Subsequent rules need only cover (and are only evaluated based on) the remaining uncovered examples, rather than the whole dataset. The process continues until all the examples are covered.

The search for each rule is organised by imposing a partial ordering among all the syntactically correct clauses and then employing a standard search algorithm. The ordering is enforced by a *traversal operator* that should be able to in-order generate all syntactically correct clauses. The ordering is either general-to-specific or specific-to-general, and the exact nature of what it means for a clause to be more general or more specific than another clause depends on the traversal operator used by each system.

Besides the structure and delineation of the search space, the other important aspect of the clause-construction algorithm is the heuristic used to guide the search, called the *evaluation function*. The evaluation function provides a measure of how good a rule is, typically based on the coverage it achieves on the remaining training data. A good evaluation function will assign higher scores to rules that are closer to the target of the search, the optimal rule, and will thus be a useful heuristic for guiding the search.

The remainder of Section 2.2 will describe various propositional rule learners and discuss their similarities and differences, thus introducing various concepts that will be used in the First-Order learners discussion later in this chapter.

2.2.2 Traversal Operators

The search space is defined, as seen above, by the traversal operator and a starting point. In systems like CN2 [15, 16] the starting point is the most general, all-accepting, empty-bodied rule and the search space is traversed in the general-to-specific direction. The traversal operator is then a specialization operator which adds a literal to the antecedents of the rule. This operation is called *subsumption* and is an incomplete form of propositional deductive inference. (See Section 2.3.2.1 for more on inference operators, including definitions.)

The *resolution rule*, on the other hand, is a complete deductive inference operator and directly corresponds to the *Modus Ponens* method of inference. In Propositional Logic the resolution rule is expressed as:

$$\textbf{Resolution:} \qquad \frac{A \rightarrow q \qquad q \wedge B \rightarrow p}{A \rightarrow q \qquad A \wedge B \rightarrow p}$$

to mean that $A \wedge B \rightarrow p$ can be resolved from $q \wedge B \rightarrow p$, provided that $A \rightarrow q$ is true; p and q are propositions and A and B are conjunctions[6] of propositions.

One can invert the resolution rule to create an operator that infers the premises from $A \wedge B \rightarrow p$. The two inductive inference rules shown in Figure 2.3 are generalizing $A \wedge B \rightarrow p$, each taking one of the original premises as a premise, to infer the other. (Remember at this point that the resolvent $A \wedge B \rightarrow p$ is more specific than either of its premises.)

[6] As the observant reader will note, a conjunction of propositions is a proposition anyway, so there is no real difference of 'type' between A, B and p, q. The distinction in the notation is there for purely conceptual reasons: the reader should think of A and B as 'programmes' and p and q as single 'clauses', in order to help transfer these concepts into Logic Programming later in this chapter.

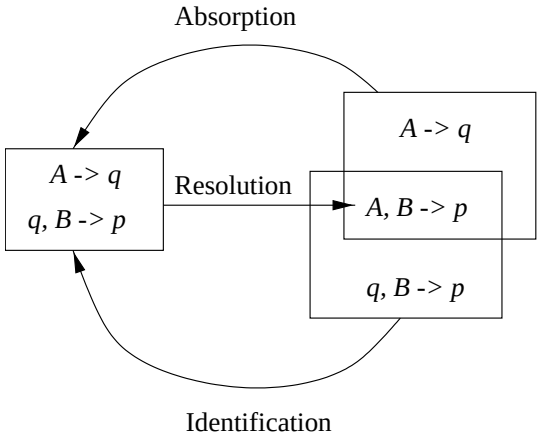


Figure 2.3: Propositional Resolution, Absorption, and Identification

Absorption generalizes $A \wedge B \rightarrow p$ to $q \wedge B \rightarrow p$, where $A \rightarrow q$ is the basis of the assumption that the (fewer) attributes of q are adequate to support p, and A can be weakened to q:

$$\textbf{Absorption:} \qquad \frac{A \rightarrow q \qquad A \wedge B \rightarrow p}{A \rightarrow q \qquad q \wedge B \rightarrow p}$$

Identification uses $A \wedge B \rightarrow p$ and $q \wedge B \rightarrow p$ to induce that q is (equivalent to) one of the atoms conjoined in A:

$$\textbf{Identification:} \qquad \frac{A \wedge B \rightarrow p \qquad q \wedge B \rightarrow p}{A \rightarrow q \qquad q \wedge B \rightarrow p}$$

Absorption and identification are collectively known as the V-operators, and they are reverting a single resolution step back to one of the premises of that resolution. The V-operators are used in the DUCE algorithm [49] as the traversal operator: the search starts from a specific example and proceeds by using the V-operators to infer increasingly general rules, until a good rule is identified.

2.2.3 Evaluating Rule Performance

The CN2 algorithm [15, 16], is a rule induction algorithm that applies the overfitting-avoidance techniques of decision-tree induction algorithms like ID3 and C4 (see Section 2.1.2 above) to the problem of individual-rule induction. The search strategy is general-to-specific and the traversal operator subsumption, as already mentioned above.

Algorithm 2.2: The CN2 FindBestRule Algorithm

LIT is the set of all possible literals. E is a set of examples.

Function FINDBESTRULE(LIT, E)
  $RS \leftarrow \{\}$
  repeat
    $RS = RS \times LIT$ {specialize all clauses in RS}
    for clauses $C_i \in RS$ **do**
      if C_i is statistically significant **and** C_i is better than $BEST$ **then**
        $BEST \leftarrow C_i$
      end if
    end for
    trim RS
  until RS is empty
  Return $BEST$

The importance of CN2 is that by incorporating these techniques in the *evaluation function* it uses to judge rules, it can handle *noisy data*. Data is said to be noisy when it is not possible to generalize[7] it into a useful theory. There are various reasons for this phenomenon, such as measurement error during data collection or mis-classification of some of the examples. One such source of noise is, for example, manually annotated linguistic corpora. These are prepared by multiple annotators who are given as detailed guidelines as possible, but who occasionally need to apply their own judgement where different annotators might take varying decisions regarding the same syntactic construct.

One more reason why it might not be possible to perfectly fit a good, non-overspecific theory to the data is that the target concept is not representable given the chosen features. Although this is a different situation than the existence of noise in the data, it has similar symptoms and is treated in the same way.

More specifically, noisy data is handled in the ID3 family by not growing the tree until it perfectly fits the data, but by growing only as long there is still a significant information gain. CN2 is, effectively, using the ID3 algorithm to construct single-branch 'trees', each corresponding to one rule, as shown in Algorithm 2.2. The FINDBESTRULE(E) function specializes clauses by appending literals in their premises. RS is limited in size, and only the best specialization are kept, effectively implementing a beam search. The search

[7] Remember that it is always possible to construct the trivial theory that perfectly fits the training data and make no generalization, but that is never interesting.

is relying on two 'external' operators on clauses that must be pre-defined: one measuring the statistical significance of a clause and one evaluating how 'good' or 'useful' a clause is.

The statistical significance testing [16, pp. 13–14] measures the distance between the distribution of examples over classes resulting from each rule choice and the distribution that would have been the result of classifying at random. It was found that including this test does not yield any improvement since it only filters out the low-coverage rules that are constructed near the end to handle the exceptional data-points that the wide-coverage rules cannot capture. As such it is only providing a bias towards shorter theories at the expense of the last bits of coverage, rather than an overall better theory.

The operator used to measure a clause's usefulness, will be called the *evaluation function*. It is, typically, used as the heuristic that guides the search for a clause that is at the core of most induction algorithms — ILP algorithms as well as propositional rule systems. Although some syntactic bias (e.g. preference for shorter clauses) is sometimes encoded into it, the evaluation function mostly refers to the *semantics* of a clause, in the sense that its most prominent arguments are the coverage the clause achieves over the training data.

In the original CN2 algorithm, the only evaluation function available was entropy (see 2.1.2 above), but was subsequently [14] modified to use the *Laplace expected accuracy*. Laplace accuracy is a special case of the m-probability estimate [11, 23]:

$$\frac{n_c + m \cdot p_0(c)}{n + m}$$

estimating the probability that a clause classifying examples as belonging to class c is accurate. n_c is the number of examples of c covered by the clause, n is the number of examples of any class covered by the clause, $p_0(c)$ is the prior probability of class c (i.e., the probability that a random example belongs to c) and m a parameter that weighs the prior probability against the observed accuracy.

The Laplace accuracy used in CN2 is the m-probability estimate for a uniform prior distribution, i.e. $p_0(c) = 1/k$, and $m = k$, where k is the number of classes. The value of m is chosen so that the two terms of the m-probability (m-weighted prior distribution and observed frequency) are in the same value range and thus are contributing equally. Substituting these values in the formula for m-probability yields the Laplace accuracy formula:

$$\frac{n_c + 1}{n + k}$$

The advantage that accuracy estimation offers over information gain is that it favours more general rules, even at the expense of misclassifying a small number of training examples. This makes it more appropriate for real-world application where the training data contains noise and inconsistencies, since it will generalize at the expense of outliers.

As an example, let us consider an experiment where examples belong to one of two classes κ_1 and κ_2 and at some node a choice needs to be made among three possible clauses C_1, C_2 and C_3 that further refine the current clause. Let us assume that the examples covered by each clause are $(980, 20)$, $(98, 2)$ and $(1, 0)$ respectively, where the $(N_{\kappa_1}, N_{\kappa_2})$ pairs are representing the number of examples of each class covered by the clause. Knowing that we are dealing with noisy data, the intuitive choice would be for C_1, since it performs as well as C_2, but seems to be making a much broader generalization. And both are preferable to C_3, which is simply re-iterating the training data.

The evaluation of these three clauses by entropy and Laplace accuracy is then as follows:

	Entropy	Laplace Acc.
C_1 (980,20)	0.098	0.979
C_2 (98,2)	0.098	0.971
C_3 (1,0)	0.000	0.667

Entropy is unable to distinguish between C_1 and C_2, since their relative frequencies are identical. Using ID3's information gain (see 2.1.2) would suffer from the exact same problem, since the entropy is weighted by the frequency of each class in the data and no distinction would have been made between 980/1000 and 98/100. Furthermore, evaluation functions that focus on minimising entropy assign absolute preference to perfectly 'pure' partitions (like the one imposed by C_3), even in the trivial cases of maximally-specific rules that cover exactly one example.

Accuracy estimation, on the other hand, is balancing between rewarding wide coverage and penalising low accuracy on the training data, the point of balance determined by the m parameter. Empirical evidence is also presented by Clark and Boswell [14, p. 155], comparing the performance of the original CN2 with that of Laplace-function-driven CN2, on noisy datasets collected from the medical domain. CN2/Laplace is shown there to out-perform CN2/Entropy with *and* without the statistical significance filter.

Another thing that needs to be noted is that these evaluation functions are different than the decision tree heuristics in a way analogous to the way rule systems are different to decision trees. That is to say, they are evaluating individual clauses without taking the rest of the theory into account, which is in accordance with the rule systems' objective of minimising or eliminating

the interaction between rules. Decision tree heuristics, on the other hand, are estimating the usefulness of testing an attribute at a particular place in the tree, typically by comparing the whole tree before and after adding the attribute test.

2.2.4 Example-Seeded Search

The sequential-cover algorithms described up to now are performing an open-ended search for each clause. That is to, say, the traversal operator is constructing increasingly longer clauses to be tried, without any reason to stop and backtrack or abandon a path, other than arbitrary (with respect to the problem) limits imposed by the user or the computer's capacity.

The AQ algorithm first appeared as early as 1969 [43] and was being developed until 1995. The important innovation of the AQ algorithm is that it introduces the concept of the *bottom clause*[8] that is constructed from an example and that serves as a boundary for the search. The bottom clause is a very long (that is, very specific) clause that constitutes the minimal generalization of a positive example. In other words, it is a very long clause that covers the example it is based on and that only.

The bottom clause is constructed by applying propositional generalization operators (the absorption and identification operators of Section 2.2.2 above) to the example. Due to the way it is constructed, the bottom clause is semantically as well as syntactically admissible; that is, it is a clause that is not only syntactically allowed, but it also covers one example, so that all its generalizations will also cover at least one example. For more information on the bottom clause (in the First-Order framework, but also applicable here) as well as a concrete example, see also Sections 2.3.2 and 2.4.1 below.

Setting the bottom clause as the limit of the search (effectively restricting the search to the power-set of the literals of the bottom clause) achieves a delimiting of the search space in a non-arbitrary way, since clauses more specific than the bottom clause would not cover any positive examples.

2.2.5 Theory-Level Post-Processing

The biggest disadvantage of the sequential-covering strategy when compared with the divide-and-conquer strategy of decision tree-learning algorithms is that the theory as a whole never gets evaluated or revised, but only the individual rules. This will sometimes lead to sub-optimal solutions in situations

[8] Called *seed* in the AQ literature.

where the best rule at some point will leave behind a more difficult to describe data-set, whereas constructing a locally less optimal rule would allow for a globally better solution.

To alleviate this problem, some of the more recent rule induction algorithms are building upon rule learning algorithms to perform theory-level post-processing. Algorithms like RIPPER [17] and C4.5RULES [68] improve an initial rule set by performing a search where deletions and revisions (RIPPER) or deletions only (C4.5RULES) of rules are tried and evaluated at the theory level and on the complete data-set. The initial theory is in the case of RIPPER constructed by IREP [26], an incremental-covering propositional rule learner, and in the case of C4.5RULES derived from a decision tree constructed by C4.5 [67].

Interesting as it may be, this line of rule system induction algorithms is not further treated here, since it is diverging from the line of algorithms that lead up to the ILP systems that are at the core of this chapter's discussion.

2.2.6 Some More Thoughts...

Most of the fundamental concepts of ILP algorithms are already in place at this point: the sequential-cover high-level strategy, the structuring of the clause search space along the general-specific axis, and the usage of a bottom clause to delineate the search at the most-specific end.

What is missing to make the leap to First-Order theory induction is exactly what makes First-Order theories more powerful than propositional ones: variables and quantifiers. First of all, the traversal operators have to be extended from propositional resolution and propositional V-operators to their First-Order counterparts, in other words First-Order resolution has to be automated and made available to the search engine. Secondly, a First-Order generalization mechanism has to be devised, so that ground terms can be generalized to ones containing variables; this will be of particular importance for First-Order bottom-clause construction.

2.3 Introducing ILP

Inductive logic programming (ILP) is the natural extension of Propositional Logic learning with first-order variables, so that ILP systems can induce first-order logic programmes.

The class of ILP algorithms that will be mostly dealt with here, is sequential-cover, single-predicate learners that employ strategies and algorithms similar

to those of propositional rule induction systems to construct an intensional definition for the relation extensionally defined by the data.

2.3.1 The Task of ILP

The constituent elements of the ILP task are: (a) the background theory, which consists of our prior theories and domain knowledge, (b) the empirical data or observations that our prior theory is unable to explain, and (c) the resulting hypothesis which explains these observations.

We start by informally interpreting the formal definition of an ILP algorithm given by Muggleton and De Raedt [53]. The task of an ILP algorithm can, then, be defined as follows:

Definition 12 *Given background theory and empirical data that satisfy the following prior requirements:*

- ***Prior Satisfiability:*** *the data must be* satisfiable *(consistent) with the background.*
- ***Prior Necessity:*** *the hypothesis needs to be* necessary*; in other words it is not possible to deduce the positive data only from the background.*

an ILP algorithm constructs a hypothesis with the following properties:

- ***Posterior Satisfiability:*** *the resulting hypothesis does not render our theoretical system inconsistent.*
- ***Posterior Sufficiency:*** *the resulting hypothesis should be* sufficient *for explaining the observations; that is, enable the deduction of all positive data.*

This definition above demonstrates a direct analogy with Popper's definition of an axiomatisation of a theoretical system [62, paragraph 16], by thinking of the background as augmented by the hypothesis as an 'axiomatisation' of the empirical data — the 'theorems' that the axiom system has to be able to deduce. Popper then suggests that a system of axioms must be:

- *free from contradiction*, or equivalently that not every arbitrarily chosen statement is deducible from it.
- *independent*, in that no axiom is deducible from the rest of the axioms.
- *sufficient* for the deduction of all the statements in the theory

- *necessary*, for the same purpose; that is, the axioms should contain no superfluous assumptions.

effectively stipulating the same conditions, except that some requirements with respect to the quality of the hypothesis are added, as it must be an economical solution as well as a sufficient one.

2.3.1.1 Formalising in Normal Semantics

This section attempts to present a more formal definition of ILP in the Logic Programming framework. For a more complete introduction to Logic Programming the reader is referred to the work of Lloyd [39] and to the logic foundations of ILP to that of Muggleton and De Raedt [53].

As already mentioned, ILP is investigating the induction of first-order predicate logic programmes. More specifically, it is investigating the induction of logic programmes expressed in Horn clauses. A Horn clause is a disjunction of any number of negated literals and up to one non-negated literal, for example:

$$h \vee \neg l_1 \vee \neg l_2 ... \vee \neg l_n$$
$$\neg l_1 \vee \neg l_2 ... \vee \neg l_n$$
$$h$$

The positive literal is called the *head* of the clause and the negative literals are collectively known as the *body*.

Horn clauses will be here represented as either sets of literals that are meant to be logically or'ed together, or Prolog clauses:

$$\begin{array}{lcl} \{h, \neg l_1, \neg l_2, \ldots \neg l_n\} & & \mathtt{h} \leftarrow \mathtt{l_1}, \mathtt{l_2}, \ldots \mathtt{l_n}. \\ \{\neg l_1, \neg l_2, \ldots \neg l_n\} & \Longleftrightarrow & \bot \leftarrow \mathtt{l_1}, \mathtt{l_2}, \ldots \mathtt{l_n}. \\ \{h\} & & \mathtt{h}. \end{array}$$

A *substitution* is a function that maps a set of variables to a set of terms or variables. We apply a substitution to a term by replacing all variables in the term with the terms or variables indicated by the mapping. We usually denote a substitution as a set of from/to pairs and we write $A\theta$ to denote the result of applying substitution θ to term A. For example, if $\theta = \{X/Z, Y/\text{socrates}\}$ and $A = \text{father\_of}(X, Y)$ then $A\theta = \text{father\_of}(Z, \text{socrates})$.

Predicates will be disjunctions of Horn clauses, where the heads are the same up to substitution, i.e. they are terms with the same functor and arity. A predicate is satisfied if one of its clauses is satisfied. *Programmes* will be conjunctions of predicates. A programme is satisfied if all of its predicates are satisfied.

Definition 12 above is formalised in the logic programming framework by Muggleton and De Raedt [53, Section 3.1, p. 635]. In the *normal semantics* — the most general case — let B, H, D^+ and D^- be any well-formed programmes. Then:

Definition 13 *Given background knowledge B and training data D^+ and D^- (positive and negative) such that the following prior requirements are satisfied:*

$$\begin{array}{ll} \textbf{\textit{(Prior Satisfiability)}} & B \wedge D^- \not\models \square \\ \textbf{\textit{(Prior Necessity)}} & B \not\models D^+ \end{array}$$

an ILP algorithm constructs a hypothesis H with the following properties:

$$\begin{array}{ll} \textbf{\textit{(Posterior Satisfiability)}} & B \wedge H \wedge D^- \not\models \square \\ \textbf{\textit{(Posterior Sufficiency)}} & B \wedge H \models D^+ \end{array}$$

if such an H exists.

Note that D^+ (the positive data) represents the observations that should be deducible from $B \wedge H$, e.g. ground examples like `dies(socrates)`. D^-, on the other hand, is a programme that (a) is consistent with the background, and (b) the constructed theory H will have to be such, that H will not disturb this consistency. In order for D^- to be interesting (that is, to be encoding an actual restriction on the solutions for H) it must be such that if a negative instance (an 'observation' that we a priori know we can never make) is satisfied, it will lead to an inconsistency. Typically it will be clauses of the form $\square \leftarrow$ `dies(zeus)` to mean that `dies(zeus)` is a negative example.

There are various other (possibly interesting) statements derivable from these requirements. All of B, H, $B \wedge H$, and D^- have to be consistent, otherwise the posterior satisfiability would not hold. D^+ follows from $B \wedge H$ (*posterior sufficiency*), so it is also consistent.

It is known that a closure under deduction of a consistent programme will also be consistent. In other words, for every consistent programme P_1, if $P_1 \models P_2$ then $P_1 \wedge P_2$ is also consistent. For $P_1 = B \wedge H$ and $P_2 = D^+$, we can deduce from $B \wedge H \not\models \square$ and posterior sufficiency that $B \wedge H \wedge D^+ \not\models \square$. This also implies that $B \wedge D^+ \not\models \square$.

Please note that this last formula only guarantees the *posterior* satisfiability of the data. In other words, and because of the 'if such H exists' clause in Definition 13, neither D^+ nor $B \wedge D^+$ are required to be consistent. What is shown by the above is that if a hypothesis was found that satisfies Definition 13, then we know that D^+ was consistent. In more practical terms, the user does not have to ensure the satisfiability of the data, but if the data is not satisfiable then the algorithm will simply fail to form a hypothesis.

It should also be noted that another common definition of the ILP task often seen in the literature is the following:

Definition 14 *Given background knowledge B and training data D, an ILP algorithm constructs a hypothesis H such that*

$$B \wedge H \models D$$

This shorter definition only specifies the posterior properties of the theory when augmented by H and does not place any prior restrictions on B and D. Under the assumption that the prior requirements on B and D are the same as for Definition 13, the relationship between the two definitions will be explored.

Since $D = D^+ \wedge D^-$, what needs to be explored is the relation between the posterior requirements of Definition 13 and $(B \wedge H \models D^+) \wedge (B \wedge H \models D^-)$, and in particular the relation between $B \wedge H \models D^-$ and $B \wedge H \wedge D^- \not\models \square$.

We will employ once again the known lemma that the closure under deduction of a consistent programme will also be consistent, this time for $B \wedge H$ and D^-:

$$\left.\begin{array}{l} B \wedge H \not\models \square \\ B \wedge H \models D^- \end{array}\right| \Rightarrow B \wedge H \wedge D^- \not\models \square$$

In the opposite direction, however, it is not the case that for any B, H, D^- conforming to the prior requirements, if $B \wedge H \wedge D^- \not\models \square$ then $B \wedge H \models D^-$. Consider, for example, $B = \mathrm{p(a)}$, $H = \mathrm{p(b)}$, and $D^- = \mathrm{p(c)}$. Their conjunction is consistent, so obviously B is also consistent with H and with D^-, but D^- is not implied by $B \wedge H$.

Definition 14 expects a bit more from H than Definitions 13 and 12: the hypothesis should not simply leave the consistency of the prior theory with respect to the negative data undisturbed, but it should also support (in conjunction with B) the negative data.

Example To demonstrate and clarify the definitions above, let us suppose an ILP experiment with the following programme encoding the prior (background) knowledge:

```
human(socrates). human(plato).
god(zeus).
```

which in this trivial example consists of ground facts only, but could also include clauses like:

```
demigod(D) ← father_of(F,D), god(F), mother_of(M,D), human(M).
demigod(D) ← father_of(F,D), human(F), mother_of(M,D), god(M).
□ ← human(X), god(X).
```

The last clause above is a particular kind of background knowledge called *prior bias*, that plays an important role in ILP and will be further explained in Section 2.3.3. Finally, our example experiment would need the two programmes representing the observations:

D^+ : `dies(socrates). dies(plato).`
D^- : $\square \leftarrow$ `dies(zeus).`

and the task of the ILP algorithm would be to construct a hypothesis that explains the data given the background, for example:

`dies(X)` $\leftarrow$ `human(X).`

2.3.1.2 Formalising in Definite Semantics

Actual ILP systems (as well as the deduction engines like Prolog that they rely on) will typically not work with unrestricted programmes, but only with programmes made up from *definite clauses*. Definite clauses are Horn clauses with *exactly* one non-negated literal, i.e. Horn clauses that are not empty-headed.

As noted by Muggleton and De Raedt [53, Section 3.1, pp. 635–6], this greatly simplifies the ILP task, since for every definite programme T there is a model $\mathcal{M}^+(T)$ (its *minimal Herbrand model*) in which all formulae are decidable and the Closed World Assumption[9] holds. In this *definite semantics* framework, the task of ILP is defined as:

Definition 15 *Given background knowledge B and training data D^+ and D^- (positive and negative), an ILP algorithm constructs a hypothesis H with the following properties:*

(Prior Satisfiability)	*All $p \in D^-$ are false in $\mathcal{M}^+(B)$*
(Prior Necessity)	*Some $p \in D^+$ are false in $\mathcal{M}^+(B)$*
(Posterior Satisfiability)	*All $p \in D^-$ are false in $\mathcal{M}^+(B \wedge H)$*
(Posterior Sufficiency)	*All $p \in D^+$ are true in $\mathcal{M}^+(B \wedge H)$*

In this setting the clauses in D^- are statements that evaluate to false, so that negative examples will be of the same form as positive ones, for example:

D^+ : `dies(socrates). dies(plato).`
D^- : `dies(zeus).`

Furthermore, it should be noted that ILP systems will usually only allow ground examples as data. This is called the *example setting.*

[9] Statements that cannot be proven true, are not true.

We saw that within the normal semantics framework bias can be handled within the logical framework of ILP by simply including in the background clauses that would lead to an inconsistency if their premises were satisfied. In the definite semantics framework, however, such clauses are not possible, but bias can be efficiently implemented as extra-logical tests the clauses have to pass before being considered as solution candidates.

2.3.2 Inverse Entailment

The core of sequential-cover rule-induction algorithms is, as already seen in the propositional case above, a clause-level search for clauses that cover some positive examples that are not explained by the current theory. The search space is the set of all the syntactically and semantically correct clauses, as defined by the prior knowledge.

In order to be able to perform a search, the space must be (at least partially) ordered by a traversal operator. It is desirable that the operator is both complete and efficient; in other words, that it can generate the entire space lying between two clauses according to the ordering it imposes, and at the same time structure the space in such a manner that interesting clauses are more likely to appear early on in the search. Efficiency, however, is typically achieved at the expense of completeness, since using 'educated guesses' to guide the search is prone to result in the operator's inability to generate certain clauses. And guaranteed completeness can be inefficient; consider, for example, the traversal operator that lexicographically enumerates the whole space, imposing a total ordering and thus guaranteeing completeness.

Sequential covering ILP algorithms typically structure the hypothesis space along the general – specific axis instead. This means that the search proceeds from a maximally general clause — the *top clause* — towards a maximally specific one — the *bottom clause*[10] — or vice versa.

2.3.2.1 Specialization and Generalization Operators

The entailment operator $\models$ is being used to express a notion of *semantic* entailment or cause-and-effect between two conjunctions of clauses. In more concrete terms, given two programmes A and B, $A \models B$ means that for every substitution θ such that $A\theta$ holds, $B\theta$ holds as well.

[10]Not to be confused with what is usually called top and bottom in logic programming, namely the empty clause and the empty predicate. There is an analogy, however, in that in the sense used here, the top is also the 'most general' clause and the bottom the 'most specific' one.

Inference rules are, on the other hand, purely *syntactic,* symbolic manipulations of logic formulae:

Definition 16 *A* deductive inference rule *is a rule that maps any conjunction of clauses A onto a conjunction of clauses B such that $A \models B$.*

Definition 17 *An* inductive inference rule *is one that maps B onto A, such that $A \models B$.*

We consider, in this case, A to be the more general theory that entails B, and possibly other theories and as well. In other words, B (and all other theories entailed by A) will be specific instances of the more general concept A. Let us consider, for example, the mortality prediction theory:

(A) if X is human, then X dies. $\wedge$
Socrates is human. $\wedge$
Plato is human.

(B) Socrates dies.

where $A \models B$, since for all substitutions of X for which A is true, (that is, $\theta_1 = \{X/\text{socrates}\}$ and $\theta_2 = \{X/\text{plato}\}$) B is also true. A is here more general in the sense that it captures the generalization that 'all humans die' and can be thus used to infer not only B, but also other propositions (that, for example, Plato dies.)

A *specialization* or *refinement operator* will, accordingly, be an inference operator mapping any clause c_1 onto a clause c_2, such that $c_1 \vdash c_2$. Such refinement operator can be used as the traversal operator in a general-to-specific search. Inversely, a *generalization operator* will be mapping a clause c_2 onto a clause c_1, such that $c_1 \vdash c_2$, and will be used in a specific-to-general search.

Deductive operators (such as the θ-subsumption and resolution operators presented here) are refinement operators, whereas their inverted counterparts are generalization operators.

2.3.2.2 θ-subsumption

θ-subsumption, introduced by Plotkin [60, 61], is one such deductive inference rule. Clause A θ-subsumes clause B if the antecedents of A are a subset of the antecedents of B, up to variable substitution:

Definition 18 *If there is a substitution θ such that $A\theta \subseteq B$, then A θ-subsumes B.*

So, for example, if

$$\begin{aligned} A &= \{\text{human}(X), \neg\text{father\_of}(Y, X), \neg\text{human}(Y)\} \\ B &= \{\text{human}(\text{socrates}), \neg\text{father\_of}(Y, \text{socrates}), \neg\text{human}(Y), \\ &\quad \neg\text{mother\_of}(Z, \text{socrates}), \neg\text{human}(Z)\} \end{aligned}$$

then $A\theta \subseteq B$ for $\theta = \{X/\text{socrates}\}$.

Using θ-subsumption as a specialization operator amounts to making a clause more specific by either adding a literal or binding a variable to a ground term. Inversely, generalization is done by dropping literals or replacing ground terms with variables. When searching in either direction (specific-to-general or general-to-specific) between a (typically very specific) clause $\{H, \neg B_1, \neg B_2, \ldots, \neg B_n\}$ and the most general clause $\{H\}$ (where H contains no ground terms) the search space is confined within the set of all possible subsets of $\{\neg B_i\}$ for the best body for the clause being constructed.

It must noted, however, that a θ-subsumption search is not complete: If A θ-subsumes B, then $A \vdash B$ as well, but the reverse is not true. Consider, for example, the following clauses:

$$\begin{aligned} A &= \{\text{human}(X), \neg\text{father\_of}(Y, X), \neg\text{human}(Y)\} \\ C &= \{\text{human}(X), \neg\text{father\_of}(Y, X), \neg\text{father\_of}(Z, Y), \text{human}(Z)\} \end{aligned}$$

Although $A \vdash C$ (by simply applying A twice), there is no variable substitution[11] θ such that $A\theta \subseteq C$.

The FOIL algorithm [66] performs an open-ended search (that is, one without a bottom clause) with θ-subsumption as its traversal operator. FOIL is the natural extension of CN2 (see 2.2.2 above) to the domain of First Order Predicate Logic: it employs the same open-ended search strategy (without a bottom clause) starting with an empty-bodied top clause and specializing it by adding literals. Effectively, it is trying a breadth-first search, adding all possible literals to the current clause, and advancing to the next 'layer' of literals once the best clause at the current length has been identified. The list of candidate-literals at each layer consists of all background predicates with all possible arguments, including variables already appearing in the body so far and new variables. At least one variable has to have already appeared.

The search heuristic is based on the notion of information gain introduced by ID3 (Section 2.1.2). The gain of adding literal L to rule R is then estimated as:

$$\text{Gain}(R, L) = t\left(\log\frac{\text{Pos}(R \cup L)}{\text{Pos}(R \cup L) + \text{Neg}(R \cup L)} - \log\frac{\text{Pos}(R)}{\text{Pos}(R) + \text{Neg}(R)}\right)$$

[11] Y/Z is forced in order to unify the `human/1` literals, so the `father_of/2` literal in A has to unify with the second one in B, but with $\{Y/Z, X/Y\}$ the clause heads differ.

where $\mathrm{Pos}(R)$ and $\mathrm{Neg}(R)$ is the coverage of rule R on the positive and negative examples, respectively. The t factor is the number of the examples covered by R that are still covered by $R \cup L$, which is rewarding clauses that are sacrificing information gain in favour of generality.

2.3.2.3 Resolution

Robinson [70] introduced the extension of the resolution rule in the domain of Logic Programming. As presented by Mitchell [47, p. 297], *Robinson's Rule* is given by the following definition:

Definition 19 *Clause R is resolved from clauses C_1 and C_2 if and only if there are literals $l_1 \in C_1$ and $l_2 \in C_2$ and substitution θ such that:*

$$\begin{aligned} l_1\theta &= \neg l_2\theta \\ R &= (C_1 - \{l_1\})\,\theta \cup (C_2 - \{l_2\})\,\theta \end{aligned}$$

By algebraically solving the equation in Definition 19 for C_2, an *inverse resolution* operator can be defined which generalizes a clause R, given the background clause C_1:

$$C_2 = (R - (C_1 - \{l_1\})\,\theta_1)\,\theta_2^{-1} \cup \left\{\neg l_1\theta_1\theta_2^{-1}\right\} \tag{2.1}$$

where $l_1 \in C_1$ and $\theta_1\theta_2$ is a factorisation of the unifying substitution θ from Definition 19, such that θ_i contains all and only substitutions involving variables from C_i. Such a factorisation is always possible and unique, because C_1 and C_2 are distinct clauses where the variables in each one are independent from those in the other.

To demonstrate the above, consider the following example, where the background knowledge $C_1 = \{\mathrm{father\_of}(\mathrm{sophroniskos}, \mathrm{socrates})\}$ will be used to generalize the clause

$$R = \{\mathrm{human}(X), \neg\mathrm{father\_of}(\mathrm{sophroniskos}, X)\}$$

In this example there is only one literal in C_1 to choose from,

$$l_1 = \mathrm{father\_of}(\mathrm{sophroniskos}, \mathrm{socrates})$$

which is, obviously, not the case where more complex clauses are involved (and the choice of l_1 will influence the 'direction' in which R will be generalized). The original unification substitution $\theta = \{Y/\mathrm{sophroniskos}\}$ is factorised as $\theta_1 = \emptyset$, $\theta_2 = \{Y/\mathrm{sophroniskos}\}$. Substituting into Equation (2.1) yields:

$$\begin{aligned} C_2 &= \{\mathrm{human}(X), \neg\mathrm{father\_of}(\mathrm{sophroniskos}, X)\}\{\mathrm{sophroniskos}/Y\} \\ &\quad \cup \{\neg\mathrm{human}(\mathrm{sophroniskos})\}\{\mathrm{sophroniskos}/Y\} \\ &= \{\mathrm{human}(X), \neg\mathrm{father\_of}(Y, X), \neg\mathrm{human}(Y)\} \end{aligned}$$

The resulting clause correctly classifies human(socrates) as a positive example, just as R did, but is more general in the sense that it is applicable to the children of all humans and not just those of Sophroniskos.

The above was the generalization operator basis of the CIGOL algorithm [52]. CIGOL follows a sequential-covering strategy, with the individual clause search proceeding in the specific-to-general direction. The starting point of the search is a ground positive example, randomly selected from the pool of uncovered positives, that is generalized by repeated application of the V-operators. The advantage is that only consistent clauses are generated, allowing for a more focused search.

Inverted resolution has, however, a very important limitation: the direction of the search is influenced by the choice of the background clause C_1 upon which the generalization is based, and not the background theory as a whole. In other words, the elementary generalization step is very uninformed in terms of the prior knowledge available. In order to identify interesting paths, multiple applications of the operator against various background clauses have to be tried and evaluated, counter-balancing the original advantage of generating valid hypotheses only and thus providing for a more focused search.

2.3.3 Prior Knowledge

In the mortality prediction example above, the solution was the following single-clause prolog predicate:

```
dies(X) :- human(X).
```

the body of which consists of a predicate provided as background knowledge. Although some ILP algorithms attempt background theory refinement or even predicate invention (see Section 2.5), the focus here will be on one that employs only the background predicates provided to construct the clauses of the target predicate. The background knowledge should, therefore, represent all the relevant facts known about the domain of the concept being learnt.

Prior domain knowledge is not, however, restricted to the set of predefined concepts available to the ILP algorithm, but extends to syntactic and semantic restrictions imposed on the clauses constructed. *Syntactic bias* operates on the form of the clauses constructed, whereas *semantic bias* deals with the semantic interpretation of the clauses' variables (for example their type or whether they are used as input or output variables for each particular literal in the clause.)

To illustrate the above, let us consider the task of inducing a simple word-structure definite-clause grammar (DCG) that would explain words like

'reiteration' and 'regeneration', but also 'reiterate' and 'regenerate', where a morpheme lexicon is provided as background knowledge:

```
% morph(InList,OutList,Cat)
morph( [re|R],      R, pref(verb) ).
morph( [iterate|R], R, verb ).
morph( [generate|R],R, verb ).
morph( [tion|R],    R, suff(verb,noun) ).
```

Note that in this representation the first two arguments are the in-list and out-list of each morpheme, so that their difference is the morpheme itself. The third argument is the morphological features of the morpheme.

There are, obviously, more than one hypotheses that satisfy the criteria of Definition 13. For example, and again using the same difference-lists representation for word rules as with lexical entries, three such hypotheses could be:

```
% Hypothesis 1
word(A,B,verb):- morph(A,B,verb).
word(A,B,Cat) :- morph(A,C,pref(Cat)), word(C,B,Cat).
word(A,B,Cat) :- word(A,C,Cat1), morph(C,B,suff(Cat1,Cat)).

% Hypothesis 2
word(A,B,Cat) :- morph(A,C,pref(Cat1)), morph(C,D,Cat1),
                 morph(D,B,suff(Cat1,Cat)).
word(A,B,Cat) :- morph(A,C,pref(Cat)), morph(C,B,Cat).

% Hypothesis 3
word(A,B,Cat) :- morph(A,B,verb).
word(A,B,Cat) :- morph(A,C,pref(Cat)), word(C,B,Cat).
word(A,B,noun):- word(A,C,verb), morph(C,B,suff(verb,noun)).
```

If the theoretical framework within which the experiment is conducted specifies, for example, a maximum branching of 2, that constitutes prior knowledge that can be encoded as a syntactic restriction (syntactic bias) that would reject the first clause of Hyp. 2. Further restrictions (e.g. right-headness) might not necessarily influence the resulting theory, but speed up the algorithm. This is the case when they identify sections of the space of possible clauses that are known in advance to evaluate poorly, so that the search can be made more efficient by excluding them altogether.

The semantic bias is also used for this same purpose, by making available to the ILP algorithm prior information regarding the semantic properties of

the background predicates as well as the target concept. In the example above, that would involve assigning incompatible types to the category variables and the morpheme list variables, so that obviously wrong clauses like:

```
word(A,B,Cat) :- morph(Cat,A,C), word(C,Cat,B).
```

are not even considered. Furthermore, variables can be specified as input or output, effectively providing as bias the 'threaded' pattern in which difference lists are employed in DCGs.

The two kinds of bias seen so far are both expressed as strict constraints imposed on the hypothesised clauses, realised in the form of yes-no decisions on the acceptability of a clause. In the general case, however, there will be multiple solutions that satisfy the requirements of the problem, as is demonstrated by hypotheses 1 and 3 of our word structure example. So an ILP algorithm needs to also be able to provide justification for the solutions it offers, besides consistency and conformance to syntactic bias. To this end, ILP algorithms allow for a *preference bias* to be stated as well. Preference bias is typically realised in the form of an *evaluation function* that provides the ILP algorithm with a 'goodness rating' for each clause. It is usually used to encode the general preference for shorter clauses or clauses that achieve wider coverage or higher prediction accuracy on the training data or any other problem-specific preference deemed useful.

In the case of hypotheses 1 and 3 here, the former can be argued to be 'better' or 'preferable' because it is more general: with the appropriate additions to the lexicon it would explain other aspects of derivational morphology besides deverbal nominalization. This preference can be realised as an evaluation function that rewards wider coverage on the training examples, operating under the assumption that their distribution is not 'misleading' so that higher coverage on the training examples implies improved performance when tested on further examples, unseen during the learning phase.

The use of background knowledge and bias constitutes one of the strongest advantages of ILP. This is of relevance to the linguistic/theoretical as well as the computational aspect of machine learning of natural language:

- Syntactic bias in general allows for the restriction of the hypothesis space within the limits of a meta-theory or theoretical framework. From the point of view of *theoretical linguistics*, this makes it possible to confine the search to one particular formalism. This is not to argue for any particular choice, only that some choice needs to be made.

- From the perspective of *natural language engineering,* ILP offers an opportunity to capitalise on linguistic knowledge in order to reduce the

computational cost of searching the hypothesis space. Many alternative learning schemes, by contrast, cannot make any use of existing knowledge.

This is, of course, not to argue that control over the feature set, constraint satisfaction and bias cannot be implemented in statistical or distributed-computation approaches to machine learning. But the qualitative difference that ILP makes, is the ability to express those in an explicit and symbolic formalism (Prolog clauses) that is considered — for reasons independent from its being employed by ILP — to be particularly suitable for knowledge representation.

2.3.4 The Evaluation Function

The evaluation function is used to measure a constructed clause's 'usefulness', or in other words an estimation of how well it would perform when queried on data unseen during the training. It should be one that achieves the 'golden section' between *overfitting* and *overgeneralizing*, while at the same time applying preference bias, as mentioned above. An overfitting theory would be one that describes the data too tightly and does not make any generalizations. At its extreme it only accepts the positive examples it was constructed from and nothing else. An overgeneralizing theory would be making unjustified generalizations and would, at the extreme, accept everything presented to it as a positive. In information retrieval terms, increased 'generality' means higher recall rates, whereas increased 'fitting' or 'specificity' means higher precision. The problem of balancing between overgeneralizing and overfitting is, then, equivalent to the recall vs. precision trade-off in information retrieval.

Evaluation functions will typically judge a clause semantically and so the most commonly used evaluation functions include simple *coverage* (positives minus negatives covered), entropy (Section 2.1.2), the m-probability estimate (Section 2.2.3), or the *Bayesian probability* that a hypothesis is correct given the data [53, p. 651].

Some evaluation functions will also refer to syntactic aspects of the hypothesised clause, implementing (for example) preference bias towards shorter or simpler clauses, in accordance with Ockham's Razor (Section 1.4). Such a function is $P - N - L + 1$, where P, N is the positive and negative coverage and L is the number of literals in the clause.

One other function that should be particularly noted is the *posonly* function [55] which facilitates positive-only learning by not referring to a clause's

negative coverage, but only its generality:

$$\text{PosOnly}(P, R, L) = \log P - C \log \frac{R + 1.0}{R_{all} + 2.0} - \frac{L}{P}$$

where $(R+1.0)/(R_{all}+2.0)$ is a Laplace-corrected estimation of the clause's generality. The estimation is made by randomly generating R_{all} examples and measuring the clause's coverage R over them. The formula is balancing between rewarding coverage (the first term) and penalising generality (the second term), so as to avoid over-general clauses in the absence of negative data. The C parameter implements preference bias towards more general or more specific clauses. The third term implements bias towards shorter clauses, unless extended coverage compensates for the length penalty.

One thing that should be noted here is the flexibility that the wide range of evaluation functions allows to ILP. And, in particular, its ability to take advantage of explicit negative data by allowing the distinction between explicit negative examples (propositions that should not be covered) and non-positives examples (propositions that are not in the set of proposition that should be covered, or propositions that are not interesting.) See also Section 5.1 below for a case where although negative data is not readily available, it is possible to separate the non-positives into 'interesting' and 'not-interesting' instances, and only use the former as negative examples, thus constructing a dataset that is of higher quality (that is, is more tightly delineating the target concept, containing examples closer to the positive-negative boundary) than the one that positive-only learning would have implicitly used.

Another way of looking at the various evaluation functions is through this classification of learning setups:

- In the strictest setup no coverage over negative data is allowed and clauses are — effectively — evaluated by positive coverage and compression. Although the evaluation function does make reference to negative coverage for the purposes of its being the search heuristic, successful clause candidates always have $N = 0$.

- The restriction on negative coverage is sometimes relaxed to allow for noisy data, in which case clauses that evaluate well in spite of their being penalised for having $N > 0$ will be accepted. Some functions, like the the m-probability estimate, also provide parameters for applying a preference bias towards stricter or more liberal clauses.

- In cases where explicit negatives are neither available nor possible to generate in a manner that is more informed than sheer chance, positive-only learning is employed. This approach most closely resembles some

regression-based machine learning disciplines in that they are both trying to guess a concept's boundaries by interpolating between positive data-points, rather than by looking for a separating line between the positive and the negative points.

2.4 The Progol Algorithm

The PROGOL [51] algorithm attempts a compromise between the efficiency of θ-subsumption and the completeness of inverse resolution, by performing a general-to-specific θ-subsumption search between the top clause and an inverse resolution-based bottom clause (see Section 2.3.2 about the top and bottom clause). The PROGOL algorithm provides for single-predicate learning without predicate invention or background knowledge revision.

At the top level the algorithm iterates through the saturation-reduction-cover removal loop shown in Algorithm 2.3, and the reduction phase is an application of the general clause construction strategy of Section 2.3.2.

In the remaining of this section each of these three stages will be further explained and demonstrated with an example. The example will be the mortality prediction task recurring through this chapter, and the PROGOL implementation employed will be ALEPH 3.[12] ALEPH [79] is a very flexible and highly configurable machine learning system that implements a number of ways to perform clause discovery. For the example in this section as well as the experiments described in subsequent chapters, ALEPH was configured to implement the PROGOL algorithm described here.

2.4.1 Saturation

During the *saturation* stage of the PROGOL algorithm the bottom clause is constructed by repeated application of the inverse resolution operator (see Section 2.3.2.3). This is done by repeatedly applying the inverse resolution operator on a positive example randomly selected from the example pool (the *seed* of the clause).

This results in a minimal generalization of the example, namely a clause which entails exactly one example — the seed. The satisfiability (consistency) property of the operator ensures that the bottom clause is consistent with respect to the background. Since a longer body (i.e., more precedents) imposes more restrictions on the variables of the head and the bottom clause is the maximally specifically clause, it is typically going to be a rather long clause.

[12] And, more precisely, the last release of Aleph 3, release date 26-2-2001.

Algorithm 2.3: The PROGOL Algorithm

while Positive examples pool not empty **do**
  Saturation:
  Pick a positive example and construct the bottom clause.
  Reduction:
  repeat
    Refine Clause
    if Refined Clause conforms with syntactic bias **then**
      Evaluate Clause
    else
      Abandon Clause
    end if
  until A good Clause is identified or the bottom clause is reached
  Cover Removal:
  Remove newly covered positive examples.
  Append the new Clause to the Theory.
end while

Semantic bias (see Section 2.3.3 above) is enforced at this stage by observing type declarations for the arguments of the target predicate as well as the background predicates. In our example we have the target predicate's mode specified by the following head mode declaration:

```
:- modeh(1, dies(+character)).
```

where the second argument specifies the forms that the predicate calls may take. `+T` arguments are bound (input) variables of type T, `-T` unbound (output) variables of type T and `#T` constants of type T.

The first argument is an upper bound of the *non-determinacy* (i.e. number of successful calls) of this particular calling form of the predicate. In other words, `dies/1` is here declared as succeeding only once when activated with a bound argument. Similarly for the background predicates:

```
:- mode(1, human(+character)).
:- mode(1, god(+character)).

:- mode(1, father_of(+character, +character)).
:- mode(*, father_of(+character, -character)).
:- mode(1, father_of(-character, +character)).
```

```
:- mode(1, mother_of(+character, +character)).
:- mode(*, mother_of(+character, -character)).
:- mode(1, mother_of(-character, +character)).
```

Multiple mode declarations denote various ways of using a predicate. The `father_of/2` predicate, for example, can be used to: (a) confirm a paternity relation, (b) retrieve all the children of a character, or (c) inquire who the father of a character is. Note that the asterisk in the second `father_of/2` mode declaration stands for 'unbounded non-determinacy'.

With the above semantic information in mind, `dies(socrates)` is minimally generalized as the following clause:

```
dies(A) :-
   human(A), father_of(B,A), mother_of(C,A), human(B),
   human(C).
```

where one should note the 'chains' of input/output variables: see how the literal `father_of(C,A)` has as arguments the variable `A` which is already bound from the head and the free variable `C` which it binds, so that is can be used as the (bound) input argument of the `human(C)` literal.

To demonstrate the way that the non-determinacy bound is used, consider adding the following entries in the database:

```
mother_of(alkmini, hercules).
mother_of(alkmini, eurystheus).
```

and saturating `dies(hercules)`:

```
dies(A) :-
   father_of(B,A), mother_of(C,A), human(C), god(B),
   father_of(B,D), mother_of(C,E), human(E), god(D),
   father_of(F,E), human(F).
```

whereas if the second mode of the `mother_of/2` predicate were changed to:

```
:- mode( 1, mother_of(+character, -character) ).
```

the saturation would wield:

```
dies(A) :-
   father_of(B,A), mother_of(C,A), human(C), god(B),
   father_of(B,D), god(D).
```

In other words, it would become impossible to represent the situation expressed by `mother_of(C,A),mother_of(C,E)`; that is, 'the mother of `A` is also the mother of `E`'.

As already mentioned, except for the matching the input/output specifications of predicates' arguments, the saturation algorithm also ensures that the same variable is used only for arguments of the same type. So far we have only used one type, `character`, but if we were to include in the background a relation involving a different kind of entities, for example the way the document in which the `character` appears was typeset, we would need to enrich the background's ontology with more types:

```
:- mode(1, typeset(+document, #typesetting_system)).
:- mode(1, typeset(+document, -typesetting_system)).

:- mode(1, appears_in(+character, #document)).
:- mode(*, appears_in(+character, -document)).

typeset(this_document, pdfTeX).
typeset(dialogues, manually).

appears_in(hercules, this_document).
appears_in(socrates, this_document).
appears_in(socrates, dialogues).
```

With the background knowledge extended in this manner, the bottom clauses constructed from `dies(socrates)` is:

```
dies(A) :-
 appears_in(A,B), appears_in(A,this_document), human(A),
 father_of(C,A), mother_of(D,A), typeset(B,manually),
 appears_in(C,this_document), appears_in(D,this_document),
 human(C), human(D).
```

One limitation that should be noted is that type matching is performed by simple string comparison and there is no notion of type subsumption.

2.4.2 Reduction

Once the bottom clause has been constructed, the PROGOL algorithm performs a general-to-specific search between the top and the bottom clause (see also Section 2.3.2 for the theoretical discussion of the search). Since the traversal operator used is a specialization operator, it is called the *refinement*

operator in the PROGOL literature. ALEPH allows for user-defined refinement operators, but it defaults to θ-subsumption. Having the bottom clause delineate the far end of the search has the advantage that θ-subsumption refinement is reduced to a shortest-to-longest search in the powerset of the literals of the bottom clause.

Furthermore, in addition to a limit for the search the bottom clause also provides the guarantee that the variables appearing in the clauses constructed during the search conforms to the type specifications of the background predicates.

The refinement operator imposes a partial ordering among the elements of the search space, making it possible to implement an admissible search through it. The search strategy can be any of the usual search strategies (depth-, breadth-, or best-first, random walks, and so on). The evaluation function (see also Section 2.3.4) is used as a heuristic for strategies like best-first that rely on one.

As an example, consider the reduction of the bottom clause derived from `dies(socrates)`:

```
dies(A) :-
   human(A), father_of(B,A), mother_of(C,A),
   human(B), human(C).
```

using a best first search strategy. The dataset consists of seven positives (six humans and the demi-god Hercules) and two negatives (gods). Since the number of positive and negative examples is imbalanced, the m-probability estimate (see Section 2.2.3) is chosen as the clause evaluation function. The m parameter is set to 1.0, to mean no bias towards more general or more specific theories.

The search starts by trying out all the single-literal clauses and evaluating them on the dataset. The (slightly reformatted) ALEPH output here shows the clauses considered, giving the with the m-probability estimate of the better clauses, and marking the 'found clauses', that is the clauses that are good enough to be a solution to the search:

```
[found clause]
dies(A) :- human(A).
Pos-Neg coverage: 6-0
m-probability estimate: 0.968

dies(A) :- father_of(B,A).
Pos-Neg coverage: 3-1
```

```
[good clause]
dies(A) :- mother_of(B,A).
Pos-Neg coverage: 3-0
m-probability estimate: 0.944
```

The 'good clauses' are those that do not cover any negatives. Clauses that do cover negative examples (like the second clause here) are immediately discarded without being evaluated.

We can also see that clauses like `dies(A):-god(A)` are not even considered, since `god(A)` is not in the bottom clause. This demonstrates the advantage of using the inverse resolution operator to create a limit for the search, when compared with the open-ended search of algorithms like FOIL (Section 2.3.2.2).

The clause `dies(A):-human(A)` other than being the best-scoring clause so far is also noted as a 'clause found'; that is, a clause that is good enough to be considered a successful result of the search. The search continues by refining this best clause:

```
[good clause]
dies(A) :- human(A), father_of(B,A).
Pos-Neg coverage: 2-0
m-probability estimate: 0.926

[good clause]
dies(A) :- human(A), mother_of(B,A).
Pos-Neg coverage: 2-0
m-probability estimate: 0.926

[good clause]
dies(A) :- human(A), father_of(B,A), human(B).
Pos-Neg coverage: 2-0
m-probability estimate: 0.926

[good clause]
dies(A) :-
   human(A), father_of(B,A), mother_of(C,A).
Pos-Neg coverage: 2-0
m-probability estimate: 0.926
```

and so on until enough[13] clauses have been visited to satisfy the algorithm

[13] According to user-specified parameters limiting clause length, maximum number of layers of new variables and maximum number of clauses visited.

that this section of the search space has been exhausted. The differences to note between this toy example and a real-world run is that the 'good' and 'found' clauses are going to be much sparser and thousands of clauses will have to be visited before one is identified. When the reduction is completed, the clause with the highest m-probability estimate is the one that is going to be added to the theory:

```
[best clause]
dies(A) :- human(A).
m-probability estimate: 0.968
```

ALEPH also allows for the handling of noisy (inconsistent) data by allowing the user to set the number of negatives that can be tolerated in a clause's coverage, either in absolute (maximum number of negatives covered) or relative (minimum accuracy) terms.

The search algorithm also allows for the disqualification of clauses or the pruning of whole sections of the search space if they do not conform with syntactic bias (see also Section 2.3.3). Assume, for example, that there is a reason to disallow in the theory clauses that refer to the father of the character being tested, so that clauses like

```
dies(A) :- father_of(F, A), human(F).
```

are disallowed, but the `father_of/2` cannot be removed altogether so that clauses like:

```
dies(A) :- mother_of(M, A), father_of(GF, M), human(GF).
```

are still possible. In that case the following pruning clause[14] would have to be included in the background:

```
prune( (Head :- Body) ) :-
       Head =.. [_,X],
       (Body =.. [','|Literals] ->
         member(F, Literals)
       ;
         F = Body),
       F =.. [father_of,_,X].
```

[14] A short note to help the reader see how the Prolog code given here works: Clause bodies are represented as a term with functor `','` and arguments the body literals, unless the body has only one literal in which case the body is simply the literal. Empty bodies are represented as `true`. The `=..` operator breaks a term down to a list containing the functor and the arguments of the term.

which is set here to activate whenever the body contains a `father_of/2` literal and its second argument is the same as that of the head. Pruning not only discards the offending clause, but also forces the search to backtrack, effectively discarding all the refinements of the clause as well.

2.4.3 Cover removal

The new clause is added to the theory and all examples covered by it are removed from the positive example pool. With each new clause the current hypothesis' coverage increases, and the saturation-reduction-cover removal cycle is repeated until some termination criterion — typically depending on the performance of the current solution — is satisfied.

Outliers that are neither covered by any clause nor can be generalized into an acceptable clause are simply appended to the theory as ground clauses.

2.5 Other Approaches to ILP

The choice of ideas and algorithms presented in the this chapter is focused on the genealogy that leads up to sequential-cover single-predicate-learning ILP algorithms and, most prominently, the ALEPH implementation of PROGOL. The reason for this is that ALEPH is the system that is used for the ILP experiments described throughout this work, but also that ALEPH is a very representative example of mainstream ILP systems, including systems like CPROGOL [51] and INDLOG [10].

The most prominent characteristics of these systems is that they are sequential-cover single-predicate-learning learners. Other approaches to ILP include systems that perform a theory-level search (rather than the clause-level search of sequential-cover algorithms) and learners that perform background knowledge revision and background predicate invention.

2.5.1 Theory-Level Search

Although ALEPH was originally the PROGOL implementation that evolved out of P-PROGOL, newer versions also include more and diverse flavours of ILP, like the theory-level search introduced with ALEPH version 3. This algorithm performs a search similar to that in the post-processing phase of some propositional rule learning systems (Section 2.2.5) and its refinement operator is performing one of four potential actions:

- Add a randomly selected legal clause to the current theory.

- Delete a clause from the current theory.
- Add a literal to one of the clauses of the current theory.
- Delete a literal from one of the clauses of the current theory.

The advantage of this approach is that it is more informed about the performance of the theory as a whole, since it is evaluating the complete theory on the complete dataset instead of only a single clause on the remaining uncovered examples. The obvious disadvantage is that the size of the search space increases by orders of magnitude, as can be easily seen by comparing the possible 'refinement' steps a theory-level search has to choose from to the single-clause refine operator.

A different alternative to sequential-covering is the divide-and-conquer technique used by Boström and Idestam-Almquist [6]. It is also a general-to-specific search operating on the theory as a whole, where on each iteration one of the following actions is taken:

- Clauses that cover only positive examples are kept in the current theory.
- Clauses that cover only negative examples are removed from the current theory.
- Clauses that cover both positive and negative examples are divided (unfolded) into a number of (more specific) clauses that put together are equivalent to the original clause.

and then the algorithm iterates, until all remaining clauses cover positive examples only. This approach to ILP can be thought of as homomorphic to decision tree induction, whereas sequential-cover is more homomorphic to propositional rule induction.

MERLIN is an ILP system implementing this algorithm and also a hybrid between sequential-cover and divide-and-conquer [6, 5].

2.5.2 Background Knowledge Revision

Single-predicate learning ILP systems construct a one-predicate hypothesis that in conjunction with the background predicates explains the data. The background predicates are assumed to be both correct and sufficient to solve the problem, and no attempt is made to revise or supplement them. There are, however, ILP systems that attempt to revise or expand the background theory. The MERLIN system mentioned above, for example, unfolds and refines the background predicates.

One other direction of research into background knowledge revision is the one exploring predicate invention, as systems like CLAUDIEN [21] and GOLEM [50, 54] do. These systems hypothesise about background predicates the existence of which would make the task easier, and append them to the background knowledge.

2.6 Why ILP?

Propositional Rule System Induction and Inductive Logic Programming suffer from the inherent difficulties of symbolic (versus numerical) computation and (especially for ILP) the size of the search space. Logic programmes are also computationally expensive to use, not only learn, since they also rely heavily on performing unification and symbolic manipulation on a large scale.

The other limitation of symbolic systems is their inability to handle numerical data. Attempts to work around this started as early as ASSISTANT [32], a spin-off of ID3 that handles numerical data and inequalities, but only by requiring from the user to segment the numerical space into ranges prior to the experiment.

Logic and ILP remain, however, the appropriate solution whenever there is an advantage to be gained by employing a symbolic, non-distributed formalism. In other words, whenever the resulting theory is not to be only quantitatively evaluated and applied, but the qualitative analysis of the results is also interesting. In such a situation Formal Logic offers both the possibility to provide an explicit meta-theoretic framework within which to search for a theory, but also a theory which offers explicit justification (that is, the Prolog proof) for each classification decision made.

Single-predicate learners like ALEPH in particular are the appropriate tool whenever the task can be formulated as the learning of a single relation or a guided series of single-relation learning sub-tasks (see Section 4.5.1 for an instance of this last case.) Chapters 4 and 5 demonstrate the application of ALEPH on two linguistic tasks, Shallow Parsing and Phonotactics respectively, linguistic theory being one of the areas where symbolic representations and Formal Logic are most heavily used.

Chapter 3

Data-Parallel ILP

Despite the constant advances in computing hardware, ILP remains a computationally expensive application with learning sessions taking many hours or even days to complete, even on the most powerful processors. A four-day experiment that had to be repeated all over again because of some minor bug in the background knowledge or a powerful server brought to the thrashing point by a memory-hungry Aleph processes has to be accepted as a fact of life, for both PhD students[1] working on ILP and their unfortunate colleagues sharing the server. This makes ILP algorithms good candidates for parallel or distributed computing, so that multiple CPUs or workstations can share the computational and data-storage load of an ILP experiment.

The Message Passing Interface (MPI) is a library specification for message passing between the nodes of a parallel machine or workstation cluster. MPI allows many and varied libraries facilitating communication between the processes involved in a parallel computation to provide a uniform interface to the programmer.

This chapter describes an extension of the Yap Prolog system with an interface to MPI libraries (Sections 3.1 and 3.2) and an adaptation of the Aleph ILP system so that it can take advantage of a parallel machine through this interface (Section 3.3). The work on extending Yap and adapting Aleph was originally presented by the author in a parallel and distributed computing workshop [37]. Section 3.4 concludes by addressing the issue of what kinds of problems this particular kind of parallelism is suitable for, although this issue is also explored in following chapters.

[1] I have myself had ILP runs like the ones described in Chapter 5 last 4 or 5 days on an HP 9000/785 server and then finish successfully, meaning that there was no infinite loop or other such bug in the background knowledge.

3.1 The Message Passing Interface

The *Message Passing Interface* (MPI) is a specification for the Application Programmer's Interface (API) to libraries that facilitate datagramme-style communication between the processors of a parallel machine or workstation cluster. What is meant by datagramme-style communication is that the information is transmitted in packets rather than through pipes, although the actual transmission is typically synchronous.

MPI should not be confused with libraries implementing parallel numerical methods, or with parallelising compilers. MPI provides the message-passing infrastructure necessary for the communication between the nodes of a parallel computation, and does not automate in any way the actual parallelisation of the code as, for example, a parallelising compiler would.

It should also be pointed out that MPI is not a library, but an API specification. The advantage of conforming to the MPI specification is that programmes can link to any MPI library without modifications, allowing for greater portability between all kinds of varied and diverse parallel architectures. In the remainder of this section, the MPI functions that are pertinent to the implementation of the MPI version of Aleph will be introduced. For a more complete description of MPI, the reader is referred to the MPI standard defined and maintained by the MPI Forum [24, 25] or user's guides to either MPI in general or some particular MPI library [27, 58].

3.1.1 Basic MPI Concepts

It was mentioned above that MPI facilitates the communication between the nodes of a parallel architecture, but it would be more accurate to say that it facilitates the communication between the processes involved in a parallel computation. In order to start a program that is using an MPI library on a parallel machine, a separate, architecture-specific mechanism — a job-queueing system, for example — has to load identical copies of the program onto each and every node. The MPI library will then provide the methods for passing messages between these processes abstracting away from the architecture of the machine, so that they can be running on the nodes of a parallel computer, the workstations of a network, or even be multiple processes running on the same processor. It is of obvious benefit to spread the processes as evenly among the available resources as possible, but that is not part of the MPI protocol: it is the queueing system's task to start the processes and enforce its queueing policy. For the remainder of this chapter, *process* and *node* will be used interchangeably to refer to a node of the computation at the abstract, MPI level, regardless of how that maps to

the actual processor nodes of the hardware.

The processes involved in a parallel computation are identified by *communicator* and *rank*. A *communicator* is a collection of nodes involved in a sub-task of the computation, and a computation might keep all the available nodes within one communicator or split them among several. MPI's goal is to facilitate fast and efficient *intra*-communicator communication, whereas *inter*-communicator communication is meant to be sparser and not as performance-critical.

Within each communicator, processes are identified by their *rank*, which is an enumeration of the processes, starting from 0. The node with rank 0 will be called the *head node*. MPI defines `MPI_COMM_WORLD` to be the global communicator that groups together all the nodes, useful for applications that do not need to group their nodes into separate communicators. Since this is case here, for the remainder of this chapter all references to rank will refer to the node by that rank in the global communicator, and the communicator argument to MPI functions will not be shown, since it would invariably be `MPI_COMM_WORLD`.

The most basic operation that MPI facilitates is the point-to-point sending and receiving of a message. A message consists of an array of data, a *type*, and a *tag*. The *type* is one of several predefined data-types supported by MPI. All the the usual C data-types, like characters, integers, and floating-point numericals, are supported. It should, however, be noted that although MPI types are stored locally according to the native binary representation for that type, all the appropriate conversions[2] are carried out when typed data is been transmitted, thus making it possible to spread a computation over heterogeneous workstation clusters.

Lastly, a message carries a numerical *tag* which can be interpreted as a message type. Messages with different tags 'live' in a different space, and a receive action must specify (by tag) which sorts of messages it should be allowed to receive; the messages carrying any other tag are to be ignored. This mechanism allows for a certain degree of asynchroneity, as communications of different 'sorts' can be kept apart without having to rely on synchronising the sender and the receiver.

With the above concepts in mind, the elementary MPI operators will have the following C declarations:

```
int MPI_Send(message, count, datatype, dest, tag)
void *message;
int count, dest, tag;
```

[2]For example between ASCII and EBCDIC characters, or 1's-complement and 2's-complement integers.

```
if(my_rank == 0) {
  MPI_Send("Hello World", 12, MPI_CHAR, 1, 0);
  MPI_Send("Hello World", 12, MPI_CHAR, 2, 0);
}
else {
  char buf[255];
  MPI_Recv(buf, 255, MPI_CHAR, 0, 0);
  puts(buf);
}
```

Figure 3.1: Example usage of `MPI_Send()` and `MPI_Recv()`

```
MPI_Datatype datatype;

int MPI_Recv(message, count, datatype, source, tag, status)
void *message;
int count, source, tag;
MPI_Datatype datatype;
MPI_Status *status;
```

where `MPI_Send()` would dispatch `count` bytes from the memory location pointed to by `message` to the node of rank `dest`. To receive the message, the recipient must issue an `MPI_Recv()` specifying: the maximum number of bytes to accept and where to place them; the source node's rank or `MPI_ANY_SOURCE`; the message's type and tag (or `MPI_ANY_TYPE` and `MPI_ANY_TAG`, respectively); and the memory location where the status of the transfer should be stored. This last `MPI_Status` structure includes information such as the actual message length, type and tag.

The C code fragment of Figure 3.1 demonstrates this simplest form of message passing, where the processes with rank 1 and 2 receive a string from the head node, and then print it out. The message is marked as being of type `MPI_CHAR` (character array) and tagged with `0` by the sender, and the receivers are also specifying that they are only accepting messages of that type and tag. The third, fourth and fifth argument of `MPI_Recv()` could have also been `MPI_ANY_TYPE`, `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, to mean that they would accept messages of any type, any tag, and from any sender respectively.

The `MPI_Send()`/`MPI_Recv()` functions described above are synchronous, in the sense that once `MPI_Send()` is called, the caller is blocked until a matching `MPI_Recv()` is issued and completed (and, of course, vice versa).

The MPI protocol also allows for asynchronous communication through the *immediate* series of point-to-point communications, which return immediately instead of blocking until the transfer is completed and can thus take advantage of hardware that facilitates memory transfers independently of the computations performed in the CPU. Since these functions have not been employed in Aleph/MPI, they will not be dealt with here any further.

3.1.2 Some More MPI Functions

The strength of the MPI specification is that it provides higher-level operators which, although possible to implement with the more basic functions described above, can be implemented more efficiently by architecture-specific code. This way parallel programmes employing MPI can at the same time be highly portable between architectures and optimised for the architecture they are running on, assuming the existence of a native MPI implementation.

One such function is the *broadcast* function, which allows one node (the *root* of the broadcast) to send a message to all other processes. Broadcasting is used when a new or updated data structure needs to be propagated through all the nodes. Broadcasting is synchronous, which is to say that *all* the nodes have to reach the point in their code where the broadcast call is made before a broadcast can be successfully completed.

Broadcasting is performed with the `MPI_Bcast()` function:

```
int MPI_Bcast(message, count, datatype, root);
void *message;
int count, root;
MPI_Datatype datatype;
```

which must be invoked by *all* processes. It should be stressed at this point that the `MPI_Bcast()` calls will block the calling process until all processes have issued a `MPI_Bcast()` and the broadcast (or at least a process's involvement in the broadcast) has been completed. As is typically the case with synchronous communication, it is the application programmer's responsibility to ensure that all the nodes will reach the `MPI_Bcast()` call at approximately the same time, so that as few CPU cycles as possible are waisted while the nodes that issued the `MPI_Bcast()` call first are waiting for the rest.

Furthermore, all the nodes of the broadcast need to know in advance which is the root of the broadcast and provide its rank as the `root` argument. It should also be noted that there is no notion of message tags, so that a matching root is the only requirement for delivering a broadcast message.

Using `MPI_Bcast()`, the code snippet of Figure 3.1 would be re-written as in Figure 3.2 (enriched with some initialisation, finalisation and environment-

```
char buf[255];

MPI_Init(&mpi_argc, &mpi_argv);
MPI_Comm_rank(&my_rank);

if(my_rank == 0) { strcpy(buf, "Hello World"); }

MPI_Bcast(buf, 12, MPI_CHAR, 0);

if(my_rank != 0) { puts(buf); }

MPI_Finalize();
```

Figure 3.2: Example usage of `MPI_Bcast()`

retrieving functions explained below). It can be seen there that all nodes make the `MPI_Bcast()` call and they all know in advance which is the root of the broadcast, the type of the message, and what size buffer will be sufficient for receiving it. The difference between the root node and the receipient nodes is that the root node populates the buffer `buf` with some data and then issues the broadcast, whereas the receipients wait for the broadcast to fill in the buffer, so that they can subsequently make use of the information found there.

The broadcast call is one of the functions that demonstrate the power of MPI: the API of the `MPI_Bcast()` call is the same no matter whether it is portably implemented as a series of send and receive operations, or some more efficient architecture-specific way; for example the message could be placed in shared memory and copied from there by all the processes in architectures that support shared memory.

Some more functions that need to be mentioned are the initialisation and finalisation functions, as well as functions that provide information about the parallel environment. `MPI_Init()` accepts as argument the process's argument vector and initialises the system with the number of processes specified there. After initialisation, `MPI_Comm_size()`, for example, can be used to retrieve the number of processes in the communicator, `MPI_Comm_rank()` to retrieve the calling node's rank, and so on. `MPI_Finalize()` is called without any arguments to terminate the computation.

Finally, one should note `MPI_Probe()` and `MPI_Get_count`:

```
int MPI_Probe(source, tag, status);
```

```
int source, tag;
MPI_Status *status;

int MPI_Get_count(status, datatype, count);
MPI_Status *status;
MPI_Datatype datatype;
int *count;
```

`MPI_Probe()` can be used to receive data of unknown size, by 'probing' the queue for information without actually extracting any data. `MPI_Probe()` populates the `status` structure with information about the head message in the queue with matching `source` and `tag`. `MPI_Probe()` will block (just like `MPI_Recv()` if there is no matching message yet.

The size of the message received, however, is not directly available as a field of the `status` structure and a call to `MPI_Get_count()` is required to extract it. `MPI_Get_count()` also expects as input the datatype according to which the message will be interpreted.

3.2 The Yap/MPI Interface

Yap is a Prolog system developed at the University of Porto and at the Federal University of Rio de Janeiro [19]. Yap is the only Prolog compiler providing the depth-limit feature required by Aleph 3.

Yap includes two mechanisms for extending the library with foreign predicates, typically written in C: one static (through libraries or object-code files linked into the bulk of Yap's code) and one dynamic (through dynamic libraries, linked at run-time). The Prolog interface to MPI described here consists of a static extension to the Yap library. This approach was chosen because MPI installations might sometimes only be available as static libraries[3], forcing the interface code to be static as well.

It should be stressed that what is being described in this chapter is not parallelising or in any way modifying any of the logical aspects of Yap, and, indeed, no changes have been made to either the abstract machine implementation or the internal database mechanism. Just like MPI itself is not a parallelising compiler but only a message-passing mechanism, a Prolog interface to MPI only provides the infrastructure for passing messages between the nodes of a parallel computation. The interface is implemented as an additional foreign library and the only changes made within the existing Yap

[3] As is the case with both the MPI implementations installed on the Linux-workstation cluster of the University of Groningen where Yap/MPI was tested.

code were are at the initialisation routine, where the `mpi_*` predicates are declared and the MPI-related command-line arguments extracted and stored so that they can be used by `mpi_open/3`.

3.2.1 Prolog Term Messages

Prolog operates at a higher level with respect to data structures than C and MPI, which means that a Prolog interface to MPI cannot let the arguments of Prolog predicates simply fall through to the C calls it is based on, but it has to re-express them as one of the MPI elementary data-types

In Prolog all data is expressable as terms, which terms are (at the level of the Prolog interface) treated uniformly regardless of whether they contain integral, real, alphanumerical, or any any other kind of data as arguments. This means that the MPI interface should be able to transmit unrestricted Prolog terms, the binary representation of which might be radically different between different machines, making it impossible to simply clone the binary term and graft it in the receiving machine's memory.

For this reason it is the string representation of terms that is transmitted, which is then parsed back into a term on the receiving node. This was achieved by reusing the code of the term parser and printer available in the I/O module of the Yap system; on the sending side terms are translated into their string representation and then transmitted as arrays of type `MPI_CHAR`. On the receiving side, they are parsed back into the internal representation and the resulting Prolog term used to bind an 'output' variable.

3.2.2 Point-to-Point Communication

The `mpi_send/3` and `mpi_receive/3` predicates implement the interface to MPI's synchronous, point-to-point `MPI_Send()` and `MPI_Recv()`. They have the following semantics:

```
   mpi_send(+Data, +Destination, +Tag)
mpi_receive(-Data, ?Source,      ?Tag)
```

where `mpi_receive/3` binds the `Data` variable to the term that was sent by `mpi_send/3` if the `Source` and `Tag` variables can be unified with those of the corresponding `mpi_send/3` activated. Figure 3.3 demonstrates the usage of `mpi_send/3` and `mpi_receive/3`; it is equivalent to the 'Hello World' fragment in Figure 3.1.

The `Data` argument in `mpi_receive/3` must be an unbound variable rather than a partially or even fully instantiated term. It would have been possible to allow this argument to be partially or fully instantiated, and then

```
greet(0):- !,
     mpi_send('Hello World', 1, 0),
     mpi_send('Hello World', 2, 0).
greet(_):-
     mpi_receive(Message, 0, 0),
     writeq(Message).

:- mpi_open(Rank, NumProc, NameProc),
     greet(Rank),
     mpi_close.
```

Figure 3.3: Example usage of `mpi_send/3` and `mpi_receive/3`

simply have the predicate fail if the argument fails to unify against the term that has been received, but that would have been misleading: once the source and tag arguments match, the message will be extracted from the message queue and only then unified with `Data`. Since there is no way to push messages back into the head of the queue, the only reasonable design choice is to always accept a message if the tag and source match, in other words require that the first argument of `mpi_receive/3` is an unbound variable.

To make this point clearer, consider the two variations of the code of Figure 3.3 shown in Figure 3.4, where the message is encapsulated in a `msg/2` term which carries a filename to output to as well as the message itself. The receiving nodes perform a simple transformation on the filename sent by the head node and then print the text to a file by that name. The (correct) code to the left accepts any term (assuming the sender and tag match) and then performs the necessary checks, whereas the code to the right incorrectly assumes that because the sent message cannot be unified with the `msg(file1,Text)` term it expects, it will not be extracted from the queue and a second attempt to receive it can be made. In order to avoid this kind of confusion, `mpi_receive/3` is implemented so as to immediately fail without calling `MPI_Recv()` if its first argument is not an unbound variable.

One thing that also needs to be noted is that the functions `MPI_Probe()` and `MPI_Get_count()` are used to retrieve the size of the message before its actual reception. In other words, a successful `mpi_receive/3` activation translates into two C-level MPI calls: one to `MPI_Probe()` and one to `MPI_Recv()`[4]. This approach was taken in order to ensure that no prior limit

[4]not counting the `MPI_Get_count()` call, which is definitely not going to require any inter-process communication.

```
greet(0):- !,
  mpi_send(msg(file1, 'Data'),
           1, 0),
  mpi_send(msg(file2, 'Data'),
           2, 0).
greet(_):-
  mpi_receive(Message, 0, 0),
  (Message = msg(file1, Text),
   File = 'file1.data'
  ;
   Message = msg(file2, Text),
   File = 'file2.data')

  open(File, write, S),
  writeq(S, Text), close(S).

:- mpi_open(Rank, _, _),
  greet(Rank),
  mpi_close.
```

```
greet(0):- !,
  mpi_send(msg(file1, 'Data'),
           1, 0),
  mpi_send(msg(file2, 'Data'),
           2, 0).
greet(_):-
  (mpi_receive(msg(file1,
                Text), 0, 0),
   File = 'file1.data'
  ;
   mpi_receive(msg(file2,
                Text), 0, 0),
   File = 'file2.data')
  open(File, write, S),
  writeq(S, Text), close(S).

:- mpi_open(Rank, _, _),
  greet(Rank),
  mpi_close.
```

Figure 3.4: Example usage of `mpi_recv/3` with uninstantiated (left) and partially instantiated (right) `Data` argument.

is set on the size of the terms that will be transmitted. This is particularly important for the application of the interface described here, since Aleph needs to transmit potentially enormous lists of examples covered between the worker nodes and the master node (see section 3.3 below). The cost of the `MPI_Probe()` call depends on the specifics of the MPI implementation used, and can range from negligible (if message-queue statistics are available locally on the node issuing the `MPI_Probe()` call) to costs of a level comparable with a full `MPI_Recv()` call.

An alternative would be to implement `mpi_send/3` and `mpi_receive/3` in a way that allows for the transmission of arbitrarily long terms while at the same time transmitting smaller terms with only one invocation of `MPI_Send()`/`MPI_Recv()`. One way of achieving this would be transmitting longer messages as chains of message packets. This requires sending some extra 'control information' along with the message (most notably whether this is the last packet or not), which could be easily accomplished by, for example, encapsulating the whole message in a term. This was not done for this prototype implementation, but might be worth doing if the MPI interface is to be used for a more varied range of applications rather than only Aleph, as is currently the case.

3.2.3 Broadcasting

A similar approach was used for the Prolog interface to `MPI_Bcast()`, except that in this case there is no tag argument, since `MPI_Bcast()` does not support message tags:

```
mpi_bcast(+Data, +Root)
mpi_bcast(-Data, +Root)
```

The first calling mode is for the root node and the second for all the receiving processes. This is, in fact, enforced by the implementation of `mpi_bcast/2` by comparing the value of `Root` with the rank of the node executing the call.

Analogously to `mpi_send/3` and `mpi_recv/3`, `mpi_bcast/2` also does not depend on the user to provide a maximum message size. To achieve this, `mpi_bcast/2` is implemented as two `MPI_Bcast()` calls, the first of which is used to transmit the length of the string representation of the actual term to be transmitted.

Finally, `mpi_open/3` and `mpi_close/0` are used to start and terminate the parallel computation. `mpi_open/3` uses the user arguments in the Yap command-line (i.e. the arguments after the `--` on the Yap command line) as MPI arguments to pass to `MPI_Init()`, and then it uses the appropriate functions to retrieve the number of processes and the current node's rank

```
greet(0):- !,
     mpi_bcast('Hello World', 0).
greet(_):-
     mpi_bcast(Message, 0),
     format(Message, []).

:- mpi_open(Rank, NumProc, NameProc),
     greet(Rank),
     mpi_close.
```

Figure 3.5: Example usage of `mpi_bcast/3`

and name, which it unifies with its three arguments. `mpi_close/0` accepts no arguments, and simply calls `MPI_Finalize()`.

To demonstrate the usage of the Prolog interface to `MPI_Bcast()`, the code fragment in Figure 3.2 is given in its Prolog equivalent in Figure 3.5.

3.3 Evaluating Clauses in Parallel

The Prolog interface to MPI libraries is then used in an extension to Aleph 3 for parallel systems. The part of the theory-construction process towards which the parallelisation effort is directed is, as already mentioned above, the evaluation of each clause constructed during the search (see Section 2.3.4 in previous chapter for more details).

The predicates within Aleph that were mostly influenced were those pertaining to loading the example files (since the examples had to be distributed among the processes) and the those implementing the example-proving mechanism itself.

3.3.1 Loading the Examples

In Aleph, the example files are read and the examples asserted in the internal database (IDB) as `example/3` terms, with arguments a unique number for each example, whether it's a positive or a negative example, and the example itself. For Aleph/MPI, the examples have to be distributed among the nodes. This is done by having the head node read the example files in, assign each example its numerical ID and construct the `example/3` terms, and transmit to the appropriate node.

One of the advantages of the parallel clause evaluation is that each node

needs keep in memory only the examples that it will be evaluating each clause, allowing for data sets that wouldn't fit in any single computer to be employed.

The disadvantage of not keeping all examples on all nodes is that the work-load might become unbalanced over the course of a learning session. This is due to the fact that covered positives get removed from the examples pool, so that only for the very first clause added to the hypothesis is it guaranteed that the work-load will be evenly balanced among the nodes. This is, in practice, not as big a problem as it might seem, since for large numbers of examples it is expected that the distribution along the nodes will remain practically even. When the example pool gets small enough that some nodes might be left with drastically less work to do, then the bulk of the computation will have been already performed and the losses will be bounded within a small fraction of the total time spent. Furthermore this only applies to the positive examples, since negative examples do not get removed from the pool and will remain balanced for the duration of the learning session.

One way to further alleviate this problem is to scramble the example files, in order to prevent any patterns emerging from the example-generation process from resulting in the removal of large numbers of *consecutive* examples by one clause. The removal of large numbers of examples is, of course, still the goal of the whole process, but we would rather have the covered examples spread as uniformly as possible among the nodes. This is currently done by distributing the examples by modulo: that is, if there are three workers, the examples would be distributed like so:

worker 0	worker 1	worker 2
ex 0	ex 1	ex 2
ex 3	ex 4	ex 5
ex 6	...	

and so on, so that removing ranges of consecutive example would equally lighten the workload of all nodes.

3.3.2 Proving the Examples

Aleph/MPI is designed so that the *master* process sequentially performs the search, requesting for each clause constructed to be evaluated by the rest of the nodes (the *workers*), and collecting and collating the results.

Besides the addition of the MPI initialisation and finalisation code and the distributing of examples among the nodes, the most important differences between Aleph and Aleph/MPI lie under the `induce/0` and `prove_cache/8` predicates.

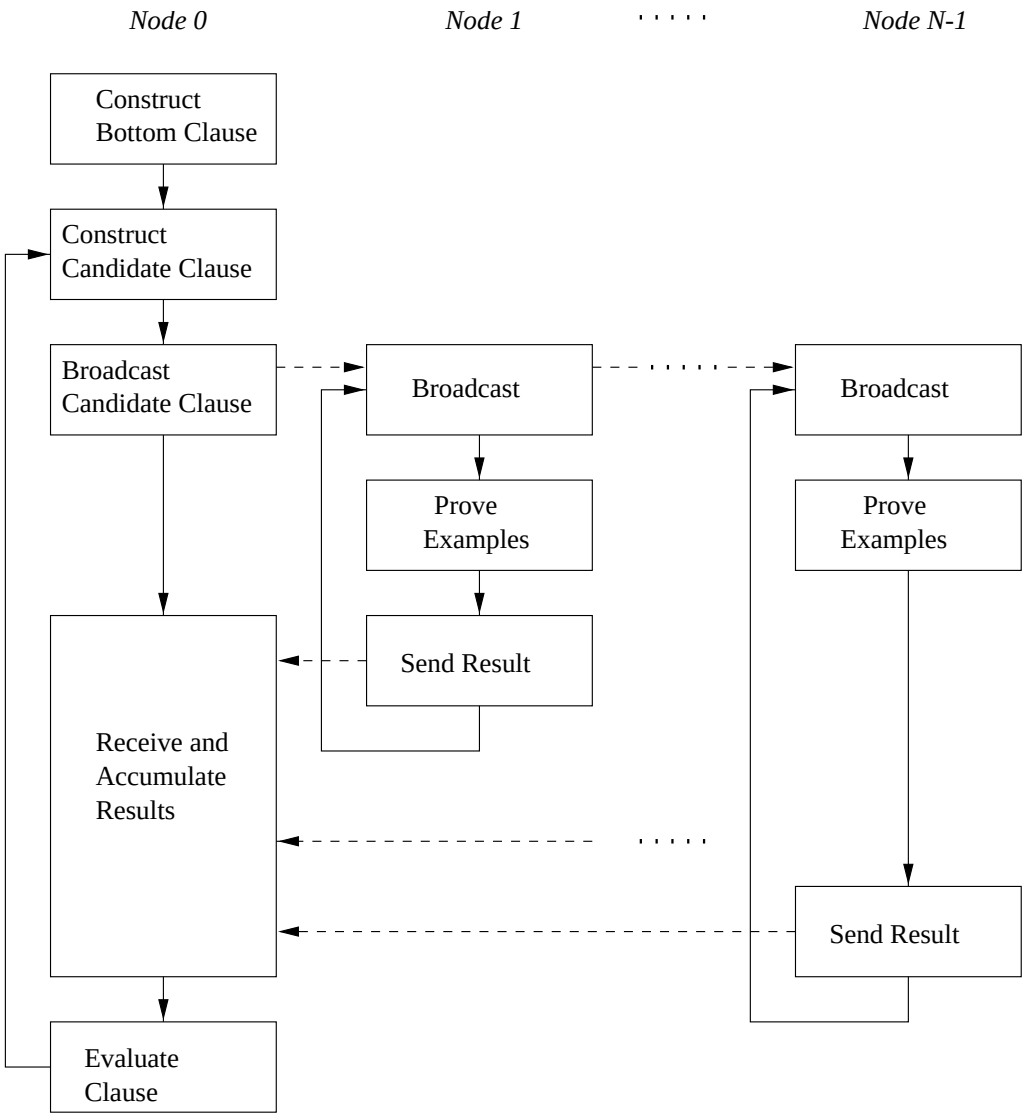


Figure 3.6: A rough schematic of the clause construction and evaluation loop: solid lines represent execution flow and dashed lines data flow between the nodes.

The `induce/0` predicate is the main clause construction and evaluation loop, supplemented in Aleph/MPI by `induce/1(+Rank)`. `induce/0` retrieves the rank of the current node and calls `induce/1`, which (as can also be seen in Figure 3.6) behaves differently for the master node than for the workers:

1. The head node (master) activates `induce/1` as `induce(0)` and has it behave, effectively, the same as the original `induce/0`, going through the example saturation, clause construction and clause evaluation loop. The difference is that the master is now broadcasting to the workers a request for the evaluation of a clause, rather than proving the clause itself. When the master receives the answers from the workers, it calculates the union of the successful example intervals and the summation of the numbers of positives and negatives covered and proceeds (as per single-processor Aleph) to apply the evaluation function based on these numbers, append successful clauses to the current theory and remove covered positive examples from the pool.

2. When activated with any non-zero rank value, `induce/1` goes into the workers' loop that issues a broadcast, acts upon prove requests as soon as they get broadcast, uses `mpi_send/3` to transmit back to the master the list of successful examples, and returns to waiting for the next broadcast.

It should be noted that it is the master's responsibility to keep track of the 'active' (not covered yet) positive examples, since it is in the master node that the decision whether to append a clause to the theory or not is made. This means that the master has to keep a list of the active example indexes (but not the examples themselves) and include this list in the prove request along with the clause that needs to be evaluated. A performance improvement could be gained by keeping this list local to the workers and issuing the appropriate update request every time a clause gets accepted into the theory, under the assumption that:

1. There will be a lot more unsuccessful clauses than successful ones.

2. There is a significantly higher cost in transmitting the examples-to-prove list, versus the cost of the update requests.

The second assumption might not be always satisfied, since it is the case that in modern workstation clusters it is the delay of establishing a connection between nodes that is responsible for the transmission costs, rather than the low bandwidth of the network.

The `prove_cache/8` predicate is the entry point to the example-proving mechanism: it first checks to see if a given clause has already been proven (and cached), and if yes returns the already calculated and cached coverage, otherwise it tries to prove the examples with this clause and returns (and caches) the results.

Aleph/MPI parallelises the example proving mechanism, so the differences between Aleph and Aleph/MPI can be hidden beneath `prove_cache/8`. In other words, the changes made to the predicates used by `prove_cache/8` as well as within `prove_cache/8` itself, are not visible to the predicates that use `prove_cache/8`, which retains the same semantics as with Aleph:

```
prove_cache(+Mode, +Settings, +Type, +Entry, +Clause,
            +IntervalsIn, -IntervalsOut, -Count)
```

unifying `IntervalsOut` and and `Count` with the number of examples within `IntervalsIn` that are covered by `Clause`. The second branch of `induce/1` uses `prove_cache/8` in the same way as `induce/0` does in Aleph, localising the changes that need to be made to introduce MPI in Aleph, and keeping them apart from the parts of Aleph that implement the search itself.

The code of Aleph's `prove_cache/8` is moved to `prove_cache_local/8`, and the new `prove_cache/8` is broadcasting the prove request to the workers and collecting the results. This latter task consists of accumulating the union of all the partial coverage list into a total coverage list, and is performed quasi-asynchronously. In particular, it is implemented as a loop of `mpi_receive/3`'s without any sender specified. As soon as a partial list is sent from any of the workers, the master calculates its union with the cover-list accumulator and reiterates to wait for the next partial result.

The workers are using `prove_cache_local/8` (containing the original `prove_cache/8` code) to do the actual proving and then transmit the list of successful intervals back to the master.

3.4 Testing Aleph/MPI

Given a programme that in some way processes or transforms its input to produce output, any attempt to parallelise it would fall under one of the two major brands of parallelism: *code-parallelism*, which distributes the work that needs to be done to process each piece of input, or *data-parallelism*, which distributes the input and processes each individual piece of input sequentially. Since ILP systems are programmes that transform an extensional definition of the target predicate (that is, the examples) to an intensional definition,

the 'input' of an ILP system is the examples and this chapter is describing a data-parallel version of Aleph.

It is immediately obvious that the choice between code-parallelism and data-parallelism is dictated by the nature of the problem to be solved: the interaction between the input data might be too dense to allow for data-parallelism or it might not be possible to parallelise the code of the process. In the case of ILP, data-parallelism is only speeding-up the evaluation of each hypothesised clause, which makes this form of parallelism suitable for tasks where a significant amount of the total computational cost is spend for clause evaluation, in other words when there are large numbers of examples available.

3.4.1 Learning the odd numbers

In order to test Aleph/MPI, a very simple induction task was devised, where Aleph had to construct a trivial theory from a large data set. This was meant to simulate a situation where the computational cost stems from the volume of the data-set rather than the size of the search space.

The task chosen was that of learning the odd numbers. The definitions of odd numbers and 'small odd' numbers were included in the background, so that the search algorithm would first try:

```
target(N) :- small_odd(N).
```

and then discover the perfect-scoring theory:

```
target(N) :- odd(N).
```

The data-set used consisted of the first 100 thousand natural numbers[5].

This setup was then used on the Beowulf Linux Cluster of the *Centre for High Performance Computing and Visualisation*[6] of the University of Groningen. This cluster consists of 96 Pentium-4 workstations with 512 Mb of memory each and 16 double-processor Pentium-4 workstations, also with 512 Mb of memory. The wall-times (as reported by the cluster's job scheduler, and averaged over ten runs) versus the number of nodes employed in the computation are given in the left column of Table 3.1.

Although the run-times shown in this table show an increase in performance up to 16 nodes, they do not even compare favourably with the performance of plain Aleph on one of the nodes of the same cluster (ten runs averaging at 139.5 seconds, std. dev. 1.3). This suggests that the time spent

[5] Obviously split down the middle between positives and negatives.

[6] See `http://www.rug.nl/hpc/` for more information

Nodes	Light		Heavy	
	avg.	σ^2	avg.	σ^2
8	991.9	63.2	9352.0	97.9
10	419.4	17.3	8773.8	1.0
12	270.3	6.5	8546.8	7.3
16	168.0	1.4	8406.3	0.6
24	145.2	1.5	8313.0	3.5
32	147.0	1.8	8281.0	3.5

Table 3.1: Wall-time performance versus number of nodes for the two `target(N):-odd(N)` experiments.

proving the examples does not outweigh the message transmission costs involved in parallelising the example-proving phase. There are two ways in which the example-proving phase might be more computationally expensive; the background knowledge might be more complex so that proving individual examples becomes more expensive, or there might be more examples.

To test the first hypothesis, the definition of the `odd/1` predicate above has been adjusted to simulate the computational cost of a heavier, more difficult to prove background:

```
odd(N) :-
  once(sleep(0.01)),
  M is N mod 2, M =:= 1.
```

and the experiment was repeated, with the new run-times shown in the right column of Table 3.1. It is immediately obvious that the heavier background has made the benefits gained from each extra processor smaller, and is still less efficient than vanilla Aleph (7165 seconds), but on the other hand the effect of Amdahl's Law manifests itself later, since with 32 nodes there is still some minor gain whereas with the original experiment adding nodes stopped paying off somewhere between 24 and 32 nodes.

The reason for this result is that a difficult background theory has an impact on the saturation stage as well as the proving stage, since the ground values encountered in the example that is being saturated are tried on the background predicates, before these predicates can be used as their minimal generalization. Consequently, 'heavier' background predicates make the learning process lengthier, but do not increase the fraction of the total time spent during the example proving phase.

3.4.2 Other Approaches

Other alternative approaches to parallelising ILP algorithms have been suggested in the literature, including employing an Or-parallel Prolog system and fully-independent data-parallelism.

An Or-parallel Prolog system can divide the search space of the ILP algorithm in 'sectors' which will be searched in parallel by the nodes of the machine. Such an approach is the one by Ohwada et al. [57] who used KL1 — a concurrent logic programming language — to implement the search component of the ILP algorithm. This approach will work best for what must the most common class of ILP applications: those where most of the time is consumed in constructing candidate clauses and traversing the search space, rather than evaluating a clause. In other words, in problems where the bottleneck is the size of the search space rather than the size of the data.

Approaches more similar to the one described here, are data-parallel ILP algorithms where it is the clause search rather than the clause evaluation phase that is parallelised. In other words, each processor constructs a clause based on its local dataset, and the partial results are then merged. The earliest such approach was by Dehaspe and Raedt [22] who implemented a parallel version of the CLAUDIEN ILP system. CLAUDIEN operates within the *non-monotonic* semantics, rather than the most commonly used *normal* semantics of PROGOL.

This latter algorithm was later tackled by Skillicorn and Wang [77], where the hypotheses constructed by each node are globally evaluated. This will perform well for disjunctive concepts, since there will be gains not only during the search itself, but also from the fact that each processor is likely to find a different (but useful) clause, so that many steps of the cover-removal strategy of Progol can be performed during one parallel step.

3.4.3 Conclusions

The most important conclusion drawn from the above is the confirmation of the fact that data-parallelism is mostly applicable to situations where the bottleneck is the volume of the data rather than the inherent computational complexity of the task. In this case it also offers the added advantage of spreading the examples among the nodes, lowering the memory requirements per node for being able to fit the examples in the machine.

There are, however, also situations where this form of parallelism is not appropriate, since the bottleneck is the size of the search space and the lack of reliable heuristics, rather than the effort of proving the examples. This is the case in experiments like the one in Chapter 5 below, where there are only few

examples available but still a large space to search. In these cases it would be preferable to parallelise the traversal of the search space, so that each node constructs and evaluates a sub-set of the clauses hypothesised before a good clause is identified. The subject of data- versus code-parallelism will be revisited in Section 5.7 below.

As far as future directions in which this work can be extended, one can start with all the technical improvements and optimisations that both the underlying Yap/MPI code as well as the Aleph/MPI code need. Then, more experimental data is needed, and in particular comparison with the closely related work of Skillicorn and Wang [77] in order to establish whether there are domains where this form of data-parallelism is advantageous over the one described by them.

Chapter 4

Shallow Parsing

This chapter deals with applying Inductive Logic Programming (ILP) to the task of *chunking*, a form of *shallow parsing* explained in Sections 4.1 and 4.2. Then an overview of previous machine-learning approaches to chunking is given (Section 4.3), while the remainder of the chapter describes the ILP chunking experiment.

4.1 Full vs. Shallow Parsing

Syntax is the study of grammar relations between words and other units within the sentence. (The Concise Oxford Dictionary of Linguistics, [41])

The syntax of an utterance is, thus, the way in which words combine to form grammatical phrases and sentences and the way in which the semantics of the individual words combine to give rise to the semantics of phrases and sentences. It is, in other words, the structure hidden behind the (flat) utterance heard and seen on the surface. At least since the early work of Chomsky [12], it is the fundamental assumption of linguistics that this structure has the form of a tree, where the terminal symbols are the actual word-forms and the non-terminal symbols abstractions of words or multi-word phrases. So, for example, the phrase 'confidence in the pound' would be assigned the structure shown in Figure 4.1 to mean that 'the' and 'pound' combine to make a noun phrase (N1) component, before combining with the preposition 'in' to form a preposition phrase (PP), and so on.

The rules according to which words and lower-level phrases of a given language may combine to form a higher-level phrase constitute the *grammar* of the language. Grammars are typically expressed as Context-Free Grammars (CFG) or in a formalism that extends CFG with feature unifications, e.g. Definite Clause Grammars (DCG), or an even stronger formalisms, such

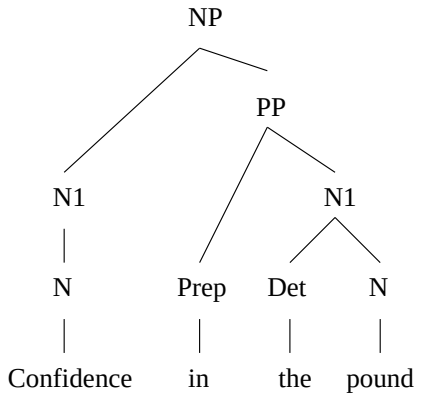


Figure 4.1: An Example Tree Phrase Structure

as Head-driven Head Structure Grammar (HPSG).[1] In the 'confidence in the pound' example here, the CFG fragment that would assign the correct structure, would look like this:

```
NP → N1 PP                Prep → in
PP → Prep N1              Det  → the
N1 → Det N                N    → confidence
N1 → N                    N    → pound
```

A DCG would also allow for features to percolate up the tree:

```
NP(Det)  → N1(Det) PP     Det(def)  → the
PP       → Prep N1(_)     Det(indef)→ a
N1(Det)  → Det(Det) N     N         → confidence
N1(none) → N              N         → pound
```

in this case marking a noun phrase as definite, indefinite, or lacking a determiner. A DCG is immediately expressible in First-Order Horn clauses (Prolog clauses) with the use of difference lists.

Parsing is the task of assigning syntactic structure to a sentence by a programme — the *parser* — implementing the grammar rules according to which words and phrases combine to form higher-order phrases. It should be noted at this point that linguistic theories differ in the constituents and rules which they postulate, and this difference must be reflected in the structures which parsers assign to strings of words. In other words, parsers must not

[1] Shieber [76] gives an overview of various unification-based grammar formalisms.

only recognise all the grammatical sentences of a language and only those, but they must also do so by building the correct structure.

Shallow parsing is the extraction of *some* features from some (speech or text) material without performing a full parse to discover its complete structure. The fragments of the structure that it does retrieve tend to be flat. As an example, consider a shallow parser that retrieves the first nominal projections (N1) of a sentence, without further analysing their contents. The full-parsing tree of Figure 4.1 would then be reduced to a tree with only two (non-terminal) levels:

(1) [Confidence] in [the pound]

where the brackets denote a sub-tree with an N1 label as its non-terminal node.

Shallow parsing is much faster than full parsing and can be very useful for tasks where the complete structure is not necessary or where the information gain from accessing the complete structure is not enough to justify the cost of full parsing. Such tasks include information retrieval, text classification or optical character recognition. In text classification, for example, it is usually enough to extract bottom-level noun phrases (here 'confidence' and 'the pound') and use them as keywords to classify the text, without paying attention to the relations among them.

Shallow parsing can also be used as a first stage for full parsing where, for example, a part-of-speech tagger might be used to assist a parser to reduce the number of parses to be considered. The information gain in this case comes for employing different paradigms for the tagger and the parser (typically a stochastic tagger and a DCG parser) so that the full parser can employ the information or relations that the stochastic model is better fitted to discover.

Stochastic models perform very well in shallow parsing tasks, and are very fast to train and apply. Furthermore — and since a shallow parser is required to assign only minimal structure to the 'parsed' material — their non-symbolic and non (or hardly) human-readable representation is less of an issue than it would have been for a full parser. In other words, in full parsing the process itself (that is, the series of rules applied) through which a sentence is recognised is important and that gives a big advantage to symbolic rule systems. In shallow parsing, on the other hand, the application of the rules is not part of the answer, which makes stochastic models more attractive.

Despite that, a symbolic, easier to manipulate representation would always be useful if it were as accurate and efficient as a stochastic one. To the array of generic (that is, not task-specific) arguments in favour of ILP

(ease of incorporating prior knowledge and enforcing a theoretical framework, human-readable resulting theories), one might add the ease of incorporating the resulting shallow parser into a bigger parsing system, since parsers' grammars are typically expressed in Horn clauses.

4.2 Chunking

Text *chunking* is a form of shallow parsing that amounts to identifying non-recursive, non-overlapping constituent *chunks* in a sentence. As an example, consider the following snippet taken from the Penn TreeBank [40]:

(2) [Confidence] in [the pound] [is widely expected] [to take] [another sharp dive] ...if [trade figures] for [September], [due] [for release] [tomorrow]...

where the bottom phrases of the full parse are shown in brackets.

4.2.1 What is a chunk?

There is no general definition of a chunk, but the general guidelines given above are used to define chunks for each language (or even task). Abney [1] introduces[2] the concept of chunks in the field of NLP and parsing, motivated from psychological as well as a phonological (prosodic) evidence presented in his paper. He defines chunks as non-recursive, non-overlapping constituents with exactly one *major head*, where a major head is a content word (as opposed to a function word) that is not between a function word and the content word it selects. To return to our pound-confidence example, 'confidence' and 'pound' are content words and they are both major heads, so that 'confidence in the pound' cannot be a chunk since it has two major heads. The function word 'the' selects for 'pound' making 'the pound' one chunk. Even if there were content-word modifiers to 'pound' (as, for example, in 'confidence in the U.K. pound') they would not be major heads and would be inside the same chunk as 'pound'.

This definition is based on English and would not be applicable in languages that would, for example, have adjectives follow the nouns they modify and not necessarily precede them as is the case in English. So, for example, in Spanish the noun phrase:

[2] Muñoz et al. [56] quote an article by Harris [28] as the one introducing the notion of chunking as early as 1957, but that is not accurate.

(3) los derechos humanos
the rights human-PL

is — intuitively — a chunk, although it includes a content word (the adjective 'humanos') that is not between a function word and the content word it selects ('los' and 'derechos', respectively).

The intuition that they are the bottom phrases of the full parse is also not an adequate definition of a chunk for various reasons:

- On the practical level — and since chunking is to be used as a preprocessing stage before full parsing — it has to be performed on plain (or at most part-of-speech-tagged) text, so it cannot be defined in terms of the full parse. And, even if it were to be defined simply as the bottom-level phrases, this would still have to be 'translated' into a language-specific definition applicable to flat text with the grammar of the language in mind.

- Chunks are sometimes defined in surprising ways due to practical considerations. Ramshaw and Marcus [69], for example, bundle the apostrophe s in phrases like 'Ramshaw's paper' in the second noun's chunk, so that this fragment would be chunked as:

  (4) [Ramshaw] ['s paper]

  While the authors provide no further motivation for this choice other that it flattens the recursive structure 'in a useful way' (Section 2.2), one can assume that preliminary experiments have shown this approach to be advantageous.

- Some syntactic phenomena might lead to surprising or counter-intuitive results if the bottom-level rule is adhered to religiously. In German, for example, one might get NPs like:

  (5) [$_{\text{NP}}$ Die [$_{\text{PP}}$ im Pfund] vertauende Leute]
  [$_{\text{NP}}$ The [$_{\text{PP}}$ in the Pound] trusting people]

  where one might argue that the most interesting or useful way to chunk the NP headed by 'Leute' is to have it include its determiner, but the base-phrase-only definition would imply that 'Leute' is a chunk by its own.

It is, therefore, the case that individual chunking experiments (sometimes even on the same language) are based on different assumptions on what an interesting chunk is and there is no general and clear definition of a chunk.

4.2.2 Noun Phrase Chunks

In analogy to phrase chunks in general, *Base Noun Phrases* (BaseNP) are bottom level, non-recursive Noun Phrases including all the NP elements up to and including the head noun. By this definition, relative clause and prepositional phrase post-modifiers are excluded and recursion is avoided. For example the Penn TreeBank snippet given above would include the following BaseNPs:

(6) [Confidence] in [the pound] is widely expected to take [another sharp dive] if [trade figures] for [September], due for [release] [tomorrow]...

BaseNPs chunking is a particularly interesting chunking task, due to the importance of Noun Phrases in typical shallow parsing tasks, such as information extraction and text classification.

4.3 Chunking as Tagging

One of the most important characteristics of the approach used to perform chunking, is the way in which it represents chunks. The most straight-forward way is bracketing, where the result is bracketed text with the restriction that brackets cannot be embedded. In Abney's paper [1] the chunker is described as a a context-free grammar used to recognise individual chunks and identify their head. These chunks would then be composed into a sentence by an *attacher*.

Abney suggests using a context-free chunk recogniser, since it is a very natural way to assign structure (that is, identify brackets and heads) to a phrase. It might, however, not be necessary to use computational machinery as heavy as a CFG to assign one-level structure, as is the case with chunking. The advantage of a CFG over a Finite State Machine (FSM)[3] is that it is able to assign more complex, multiple-layered structure to the strings it recognises, as opposed to the single layer of output of a finite state transducer. It is immediately obvious that this advantage is of no interest to the task of chunking, since we are only interested in a single layer of structure anyway.

This suggests that a more computationally efficient way to approach chunking is by representing chunks not as a one-level tree structure over the phrase, but as a *syntactic tag* associated with each word. These tags can be either open/close bracket tags, or inside/outside tags.

[3] In the cases where CFGs are used to analyse regular languages. This whole discussion is obviously not pertinent to the analysis of context free — or more complex — languages that are not representable by FSMs in the first place.

4.3.1 Bracket Tagging

Brackets can be seen as *tags* that mark the words at the beginning and at the end of each chunk:

(7) Confidence/B in/E the/B pound is/E widely expected to take another/B

so that words can be marked as opening a chunk (that is, being immediately after the opening bracket) or closing it (being immediately after the closing bracket). Adjacent chunks have to be appropriately treated, by either allowing double tagging or introducing some special 'E+B' tag:

(8) ... due for release/B tomorrow/E+B ...

One such system is described by Muñoz et al. [56, Section 3.3], where two independent predictors (in fact, networks of linear predictors) are trained to assign opening and closing 'tag candidates' with their associated confidence levels. A second pass over the tagged sentence is finding the consistent bracketing with the highest overall confidence level. A similar approach is described by Tjong Kim Sang and Veenstra [84] where the learner employed is TiMBL (a memory-based learner [30]), except that its less sophisticated bracket matcher simply discards all inconsistent brackets, aiming for precision against recall.

4.3.2 Inside/Outside Tagging

The alternative approach is to make use of the fact that each word can belong to at most one chunk (since they neither overlap nor are embedded within each other) and tag each and every word instead of only the ones at the edges of the chunks:

(9) Confidence/B in/E the/B pound/I is/E widely/O expected/O to/O take/O another/B

where words are marked as being inside (I) or outside (O) a chunk as well as opening or closing one. Words tagged as B are implicitly also marked as being inside a chunk, and words tagged as E as being outside all chunks. In fact, one could retrieve the chunks by simply marking the words with I/O tags, since the edges will be obvious:

(10) Confidence/I in/O the/I pound/I is/O widely/O expected/O to/O take/O another/I

and only using B tags to separate adjacent chunks:

(11) ... due for release/I tomorrow/B ...

This last tagging schema was introduced by Ramshaw and Marcus [69] in order to apply Transformation-Based Error-Driven Learning — a machine learning technique originally used by Brill [8] to construct part-of-speech taggers — to the problem of chunking.

4.3.3 Comparison

Tjong Kim Sang and Veenstra [84] and Muñoz et al. [56] have compared the two approaches by using the same machine learning technique to learn taggers for each of the two tagging schemes and compare the results.

Qualitatively, the main difference between the two approaches to syntactic tagging described above have to do with the consistency of the resulting tagging. Bracketing is more prone to result in inconsistent assignments (i.e. unbalanced brackets) and thus requires more sophisticated post-processing to pick the brackets from among the 'bracket candidates' proposed by the tagger. Inside/Outside tagging, on the other hand, is more robust, since all possible taggings are valid.

The quantitative results are expressed in terms of *precision*, *recall*, and F_β-*score*, which are the metrics typically used in information retrieval and machine learning. *Precision* is the ratio of true positive predictions over all positive predictions. High precision means that the model is not too liberal with accepting an example as positive. *Recall*, on the other hand, is the percentage of the positives that are predicted as such. High recall means that the model is not too conservative about accepting examples. The F_β-*score* is defined as

$$F_\beta(P, R) = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

where P is precision, R is recall, and β is the parameter balancing the importance of the two.

The three metrics above, are then used to quantitatively compare the two tagging approaches. Both comparisons seem to suggest that the two approaches are equivalent. Muñoz et al. [56] report recall of 92.5% at 92.2% precision ($F_{\beta=1} = 92.4$) for Inside/Outside tagging and 93.1% recall, 92.4% precision ($F_{\beta=1} = 92.8$) for bracket tagging. In the case of the memory-based learner [84, Table 6], bracket tagging yielded 90.8% recall at 93.7% precision ($F_{\beta=1} = 92.2$) and I/O tagging 92.3% recall at 92.5% precision ($F_{\beta=1} = 92.4$)

One important factor is, of course, the correlation between invalid taggings and wrong taggings. In other words, in the case of bracket taggers for example, a very strict bracket-matching scheme is going to improve precision, since the chunker is taking fewer 'risks' when making a positive decision. The deciding factor on whether this is going to improve performance is the price paid in terms of recall, since the more security the chunker requires before answering positively, the fewer positive answers there will be.

This drop in recall will be lower if invalid bracketings are more likely to also be wrong bracketings, in which case strict bracket matching has a positive side-effect. Tjong Kim Sang and Veenstra [84, Table 6] compare three bracketing schemes where one is stricter than the other two: the first assigns a bracketing only in the presence of an opening and a closing bracket of matching phrase type, whereas the other two allow mismatches. The quantitative results show that the precision gain does indeed balance the recall drop, to the effect that the F-score is in all three cases around 92%.

4.4 Inducing a BaseNP Chunker

The task of automatically constructing a chunker has been reformulated in the previous section as one of constructing a syntactic tagger, marking each word with a tag denoting the kind of chunk it is in or marking it as not being in any chunk. The basic motivation for this is that this way the task is formulated like a transduction task rather than a parsing task, reducing its complexity from that of a context-free language to that of a regular language. In other words, the most complex part of the space of all possible tagging schemes has been excluded from our set of potential tagging schemes and in exchange it is possible to implement the tagger with a FSM instead of more complex computational machinery. This has a significant advantage since simpler machines are easier to learn as well as more efficient to apply.

It does, then, beg the question why would one apply a Horn clause learning method like ILP to a task that can be tackled with much simpler FSM-learning approaches. It can argued however, that it will be interesting to experiment with inducing a chunker, for the sake of the formalism itself rather than the formalism's descriptive power. In other words, it might be interesting to try the more complex mechanism because of the brevity, intuitiveness, or readability of the rules and despite the fact that the language described by these rules is not complex enough to necessitate their usage. In many respects this is analogous to the advantages of using a very powerful grammar formalism like HPSG to describe natural language syntax, although there is barely an argument for its being even context free.

The chunker will nevertheless be formulated as a syntactic tagger, as described above, and not as a CFG. In other words the descriptive power that Horn clauses offer will be focused not on the structure assigned on the phrase being parsed, which will be minimal, but on the justification (that is, formal logic proof) provided for each chunking decision made.

Learning such a tagger can be fitted in the context of a single-predicate learning ILP system that does not perform predicate invention or background knowledge refinement: the target predicate is the relation between a word and its syntactic tag, and the examples can be easily extracted from a parsed or chunked corpus. This section describes employing such an ILP system (Aleph, see Section 2.4 above) to learn a BaseNP chunker from the corpus used by Ramshaw and Marcus [69], a derivative itself of the Penn TreeBank [40]. The work on shallow parsing presented here was originally published in the Proceedings of the Computational Linguistics in the Netherlands 1999 conference [33], and follow-up work in the proceedings of the 2002 edition of the same conference [36].

4.4.1 Experimental Setup

Based on the results described in Section 4.3 above, the Inside/Outside tagging scheme has been chosen, since (a) there appears to be no performance advantage in choosing bracket tagging, and (b) Inside/Outside tagging has the advantage that it requires no post-processing. This is especially the case with a machine learner like Aleph, where there is no probability assigned to each tagging decision made, making it more difficult to apply any informed bracket balancing scheme.

The target concept (the syntactic tagger) is represented as a `tagger/4` predicate that relates a word and its context to a syntactic tag:

```
tagger/4 (+LeftContext, +Word, +RightContext, ?SyntacticTag)
```

where the input arguments are the word to be syntactically tagged and its context. The tagger is meant to be used in a left-to-right pass over a sentence that has already been part-of-speech-tagged, so that the left context carries syntactic tags as well as part-of-speech tags, where the right context holds part-of-speech tags only.

The contexts are Prolog lists of word terms, and each such term encapsulates the word-form, the part-of-speech tag, and — where applicable — the syntactic tag. The word to be tagged is also a word term. Examples of these two kinds of word terms are:

```
w(confidence,nn,inp).
w(widely,rb).
```

where the two part-of-speech tags stand for 'common noun, singular' and 'adverb', respectively. The tag-set used here is the one in the Penn TreeBank — from which the dataset was extracted — described at length by Santorini [72]. The syntactic tag is simply one of `bnp`, `inp` or `o`, as explained above.

To further demonstrate how the syntactic tagger works, consider the sterling-confidence example used above. The part-of-speech tagger would tag the beginning of that sentence as:

(12) Confidence/nn in/in the/dt pound/nn is/vbz widely/rb expected/vbn to/to take/vb another/dt ...

When the tagger is applied to the first word it would be activated as:

```
tagger([], word(confidence,nn),
       [word(in,in), word(the,dt),...], Tag)
```

and would unify `Tag` with `bnp`. For the second word,

```
tagger([word(confidence,nn,bnp)], word(in,in),
       [word(the,dt), ... ], Tag)
```

The tagger would unify `Tag` with `o`, and so on until the the whole sentence has been syntactically tagged.

The relation that this predicate is defining is meant to be a function, so that no backtracking is performed and the tagging is performed as a single, tractable pass over the sentence. This cannot be guaranteed by the learning process, in the sense that multiple activations of the learnt predicate are possible. This can be tackled in various ways; here a first-match-wins policy is enforced by the programme that employs the `tagger/4` predicate to produce tagged text.

4.4.2 The Dataset

Given the above, one example can be constructed from each word in the dataset:

```
tagger([word(confidence,nn,bnp)],
       word(in,in),
       [word(the,dt), word(pound,nn), ... ],
       o)
```

Negative data is constructed by simply flipping the syntactic tag in a positive example. For this purpose, `bnp` and `inp` tags are taken to be in the same 'class' of tags, i.e. the class of tags that mark words inside a BaseNP.

Substituting one for the other might generate too many false negatives, which is avoided by always choosing a tag from a different class to generate a negative example with. When flipping an `o` tag the choice between `bnp` and `inp` is made so that the result is a valid tagging, i.e. no `inp` tag is put immediately after an `o` tag. This way the positive example above would yield the following negative one:

```
tagger([word(confidence,nn,bnp)],
       word(in,in),
       [word(the,dt), word(pound,nn), ... ],
       bnp)
```

The implications of this way of generating negative data is that there are no examples of inconsistent (with respect to the the tagging scheme) tagging in the negative data, but only examples of wrong tagging. In other words, the clauses constructed by the ILP system do not need to ensure that there will be no `inp` tag immediately following an `o` tag, since that can be easily checked and fixed on the tagged text.

4.4.3 List-Access Background Predicates

A general, theoretic overview of background knowledge has been given in Section 2.3.3, so this section will concentrate more on the way in which it has been employed for this experiment.

The two arguments of the `tagger/4` predicate that specify the context are lists of word terms, so the background predicates must include methods for accessing and manipulating lists. There are two ways to look at a Prolog list; either as a random-access array the members of which can be accessed by their offset from the beginning of the array or as a linked list where each element is pointing to the next element in the list.

Random access is implemented through the `nth/3` predicate[4] which is matching a list and an offset with the item found at that offset in the list. The list access methods built upon this predicate are `get_ptag/3`, `get_stag/3` and `get_wform/3` which retrieve the part-of-speech tag, syntactic tag and word form at the given offset, respectively.

[4] Traditionally, Prolog lists are linked lists with random access implemented by recursively traversing the list. The Yap compiler implements random access arrays as well, but for the sake of demonstration linked lists will be used for both cases. If, however, the syntactic tagger were to make many references to items deep in the list, the lists would have to be substituted with arrays for efficiency. The conversion is trivial, so this has no impact on the experiment as a whole.

Linked list access is viewing the list as a head element and a tail list. The tail is a linked list itself, also consisting of a head and a tail, and so on through the list until the last element that has an empty list as a tail. This way the first element of list L is referred to as head(L), the second as head(tail(L)), the third as head(tail(tail(L))) and so on. Prolog provides the `head/2` and `rest/2` relations, so that the above would be formulated as:

```
head(L, FirstElement).
rest(L, Rest), head(Rest, SecondElement).
rest(L, Rest1), rest(Rest1, Rest2), head(Rest2, ThirdElement).
```

Building upon the `head/2` predicate, the linked-list access methods consist of the `head_pos/2`, `head_synt/2` and `head_wform/2` predicates, that retrieve the part-of-speech tag, syntactic tag or word form of the head element, respectively. The `rest/2` predicate is made available as is. It should be noted that the left context list is reversed, so that its head is the word closest to the focus word. So, for example, the example extracted from the third word of the 'confidence in the pound' sentence would be:

```
tagger([word(in,in,inp), word(confidence,nn,bnp)],
       w(the,dt),
       [word(pound,nn),word(is,vbz), ... ],
       bnp).
```

This approach (when compared to random-access predicates) constitutes implicit preference bias towards shorter dependencies, since the search is in the general-to-specific direction. This means that shorter (more general) clauses will the visited first, and the shortest clause that is 'good enough' (according to the evaluation function) will be chosen. It also imposes a limit on the longest dependency, since there is a limit on the length of 'chains' of new[5] variables. The following clause, for example:

```
tagger(A, word(nn,_), _, inp) :-
   rest(A,B), rest(B,C), C=word(dt,_,b).
```

is referring to a determiner two positions to the left, and introduces a 'chain' of input-output variables to do so. In the experiment described here, a maximum of 7 layers of new variables is set, which imposes a prior limit on the word distance within the text to which a rule can refer. There is no prior motivation for picking any particular value for this limit, so it should be seen as a working assumption the validity of which needs to be confirmed

[5]Variables not appearing on the head, but introduced in the body by the learner.

at the end of the experiment by examining the longest `rest-head` chains that appear in the rules. It should be stressed however that allowing for longer chains has a dramatic effect on the size of the bottom clause and, subsequently, that of the search space.

4.4.4 List-Manipulation Background Predicates

Besides access to the elements of the context lists, the background includes the list manipulation predicates provided in the Prolog library, like `reverse/2` and `member/2`. Furthermore, the `current_phrase/2(+Context,-Phrase)` predicate is provided, matching a context list with the BaseNP up to this point (or an empty list if the last context word is marked as being outside a BaseNP. This predicate is only applicable to left context lists, since it is using the syntactic tags already assigned to extract the BaseNP.

The resulting sub-lists can then be accessed in the same manner as the full lists themselves. Potentially interesting pieces of information that can be extracted in this way are, for example, whether the BaseNP under consideration is definite or not:

```
is_definite_NP(Context) :-
  current_phrase(Context, A), reverse(A, B), head(B, the).
```

or whether there is already an adjective in the BaseNP or not:

```
has_adjective(Context) :-
  current_phrase(Context, A), member(w(jj,_,_), A).
```

4.4.5 Linguistic Background

Some part-of-speech tags carry some morphological and syntactic information as well as the word's part-of-speech in the strict sense of a lexical class. Nouns, for example, are tagged with one of four possible tags: singular common noun (nn), plural common noun (nns), singular proper noun (nnp) or plural proper noun (nnps). This lends itself to a classification based on two plus-minus features, PLURAL and PROPER. This information can be very simply encoded in the following relations:

```
tag_plur(nn,  -). tag_prop(nn,  -).
tag_plur(nns, +). tag_prop(nns, -).
tag_plur(nnp, -). tag_prop(nnp, +).
tag_plur(nnps,+). tag_prop(nnps,+).
```

Part of Speech	Syntactic Tag	Part of Speech	Syntactic Tag
determiners	bnp	nouns	inp
wh-determiners	bnp	adjectives	inp
existential 'there'	bnp	comparative adj.	bnp
pre-determiners	bnp	superlative adj.	inp
apostrophe-s	bnp	cardinals	inp
pronouns	bnp	foreign words	inp
possessive pronouns	bnp	symbols ($, &, etc)	bnp
wh-pronoun	bnp		
possessive wh-pronoun	bnp		

Table 4.1: The Baseline PoS to Syntactic Tag Map

making it possible to refer to all possible subsets of noun tag. Adjectival and verbal tags are also broken down into similar morphosyntactic features.

Determiners are marked with a single part-of-speech tag and no distinction is made between singular and plural, since no such distinction is made in the original corpus either.

Furthermore it might be useful to be able to refer to any noun tag, or any nominal (noun or adjective) tag, etc:

```
tag_nominal(nn,  +). tag_verbal(nn,  -). % nouns (sg)
tag_nominal(nns, +). tag_verbal(nns, -). % nouns (pl)
tag_nominal(jj,  +). tag_verbal(jj,  +). % adjectives
tag_nominal(prp, -). tag_verbal(prp, -). % prepositions
```

and so on, covering all the tags that are pertinent to any of the four major lexical categories (nouns, verbs, adjectives and prepositions).

4.4.6 The Baseline Theory

A 'naïve' tagger can be easily derived from the training set, by simply matching each part-of-speech tag to its most likely syntactic tag. For this particular experiment, the part-of-speech tags that were matched against `bnp` or `inp` tags are listed in table 4.1, with the remaining tags being marked as being outside a BaseNP, including part-of-speech tags that might appear in the test data but were not encountered in the training data.

When used as the baseline theory it scores remarkably well, especially with respect to accuracy, (see results, below) which suggests that it is encoding interesting information that could be useful to the learner.

The baseline tagger is included in the background as the `naive/2` predicate, matching each part-of-speech tag against its most probable syntactic tag:

```
%% naive/2 (?PoSTag, ?SyntTag)
naive(nn, inp).
naive(dt, bnp).
```

and so on.

4.4.7 Prior Bias

The semantic bias is declared with *mode declarations* that specify the manner in which a predicate is meant to be used. The information each predicate's mode carries is its arguments' mode, type, and non-determinacy. *Mode* specifies a term as being an input variable, output variable or ground term. The *type* is a label each variable bears and much match the type of an argument before the variable is considered as a value for that argument. The *non-determinacy* of a predicate sets an upper bound on the number of times it can succeed.

The `tagger/4` predicate is declared as:

```
:- mode(1, tagger(+wslist,w(+pos,+word),+wlist,-stag)).
```

The first argument is an upper bound of the *non-determinacy* (i.e. number of successful calls) of this particular calling form of the predicate. The second argument is specifying a form that the predicate calls may take. `+T` arguments are input variables of type T and `-T` output variables of type T.

A determinate predicate would be one that can succeed in one way only, for example:

```
:- mode(1, head_synt(+wslist,-stag)).
:- mode(1, head_pos(+wslist,-pos)).
:- mode(1, head_pos(+wlist, -pos)).
```

These examples also demonstrate how the variable types are being used to restrict the applicability of `head_synt/2` the left context only, since the the right context has not been syntactically tagged yet.

4.5 Results and Conclusions

The setup described above was used to train on a data set of 6338 positive and 6337 negative examples. The evaluation function used was Laplace estimated accuracy (see Section 2.2.3).

The resulting tagger consisted of 160 clauses, 11 of which constitute a substantial generalization and cover the vast majority of the positive examples. The remaining 149 are ground clauses that are simply verbatim re-iterating the outlying positive examples that could not be generalized in any useful way.

The constructed theory was then tested by syntactically tagging 2012 unseen sentences and calculating the BaseNP precision and recall rate of the syntactic tagger. The theory achieved a recall rate of 85.32% with 78.62% precision, improving the 75.38% recall with 75.01% precision of the 'naïve' theory taken as the baseline. (See Section 6.2 for comparison with similar work.)

One thing to be noted about these results is that perfect part-of-speech tagging is assumed, which will generally not be the case. More moderate results are expected against input pre-processed through a part-of-speech tagger, rather than input extracted from the part-of-speech tagged corpus.

Regarding the more qualitative aspects of the resulting theory, some of the constructed rules are reasonable and intuitive, whereas others are not. One commonly recurring pattern is the conditional use of the `naive/2` predicate, so that the constructed theory is effectively specifying the contexts in which `naive/2` is correct and limiting its application to those cases.

Some of the most convoluted rules of this kind look like this one here:

```
tagger(A,w(B,C),D,E) :-
   head_pos(D,F), head_synt(A,G), rest(D,H), head_pos(H,F),
   rest(H,I), head_pos(I,J), naive(J,G), naive(B,E).
%% [laplace estimate] [0.95122]
```

which stipulates that the syntactic tag `E` should be what `naive/2` predicts, given that:

- `head_pos(D,F),rest(D,H),head_pos(H,F)`: the part-of-speech-tag of the two first words to the right is the same, no matter what it is.
- `head_synt(A,G),rest(H,I),head_pos(I,J),naive(J,G)`: the syntactic tag of the first word to the left is the same as the tag predicted by `naive/2` for the third word to the right, no matter what it is.

This rule (and others like it) are an example of a theory that is not representable in a formalism weaker than Horn clauses, due to its usage of variables. Although it is always possible to unroll such rules into series of ground rules (in the same way that in finite domains Definite Clause Grammars can be re-written as longer CFGs), Horn clauses are more concise and readable. This last observation doesn't, however, mean that all the rules

are necessarily intuitive or interesting, only that the formalism allows for potentially interesting rules.

One problem to be noted with the experiment conducted is the absence of syntactic bias. It is difficult to specify syntactic bias because of the way the data is represented: by breaking up the sentence bracketing task into that of tagging individual words, the theory constructed is, in a way, 'distributed'. In other words, it is not easy to identify clearly the role of each clause in identifying a BaseNP, and the bracketing is the effect of the interaction between clauses rather than the result of the application of the appropriate clause for each particular case.

From the above it is clear that rules enforcing a theoretical framework such as, for example, X-bar theory's 'each XP must include a head X' cannot be easily represented as syntactic bias in the current setup. In general, this setup has difficulties with rules that are not local, either horizontally (long-distance dependencies) or vertically (that is, ones that make reference to complex tree structures like X-bar theory does).

As it has already been argued in Section 2.3.3 however, one of the strongest points of ILP is that it provides an intuitive formalism for declaring syntactic bias; so if it is not possible to take advantage of it, the motivation for choosing ILP for the task is undermined. Further experiments on the task of BaseNP chunking need to be focused on defining more complex and more linguistically informed background theories within this formulation of the problem, as well as devising other formulations that might be better fitted to the ILP framework.

A related issue is that the theory is not as human-readable as one might expect for a logic programme, making the task of qualitatively evaluating the result much more difficult that it would be if the theory was a Definite Clause Grammar (DCG) or some other more intuitively appealing formalism. Work in this direction would involve developing tools to extract the information in a theory thus constructed and reformulated in a more human-readable form.

4.5.1 Cascades of Chunkers

The other important limitation encountered stems not from ILP, but from syntactic tagging itself. Syntactic tagging as it stands, can only partition a sentence into non-overlapping chunks, thus making recursive theories unrepresentable. This renders the technique inapplicable to the task of top-level NP identification, since top-level NPs will inevitably contain smaller ones in preposition phrases or relative clauses.

One way in which recursion could be mimicked would be to break up the task in a bottom-up fashion. A separate tagger would then be induced

for each layer and parsing would proceed by repeated steps of recognising constituents and replacing them with a head node symbol. For example consider our original example,

(13) [[Confidence] in [the pound]] is widely expected to take [another sharp dive] if [[trade figures] for [September]]...

where higher-level NP bracketing is also marked. This could be syntactically tagged in two stages, the first one of which would be BaseNP chunking. Then all BaseNPs found are replaced by a label and in the new tagging problem these labels are treated as nouns:

(14) [BaseNP/nn in BaseNP/nn] is widely expected to take BaseNP/nn if [BaseNP/nns for BaseNP/nnp]...

This approach, however, suffers from data sparseness at the higher levels, as well as from the fact that erroneous decisions made at the lower levels cannot be revisited as would be the case with a backtracking parser. In other words, by the time the tagging layers have reached the level of top-level NPs, the results will not be reliable enough to be useful for information extraction.

A different way of using such a cascade of chunkers is as a heuristic for a full parser [7], in a setup where the chunker's suggestions are corroborated by the full parser and lower-level chunkings can be backtracked out of, if proven wrong at higher level. Such a setup, however, is restricting the shallow parser's domain of applicability to that of a pre-processor (or, rather, co-processor) for the full parser, leaving out tasks where a full parse is not necessary.

Chapter 5

Phonotactics

The *Phonotactics* of a given language is the set of rules that identifies what sequences of phonemes constitute a possible word in that language. The problem can be broken down to the *syllable structure* (i.e. what sequences of phonemes constitute a possible syllable) and the processes that take place at the syllable boundaries (e.g. assimilation).

Previous work on the syllable structure of Dutch includes hand-crafted models, like the ones described by van der Hulst [87] and Booij [4], but also machine-learning approaches: abduction by Tjong Kim Sang and Nerbonne [83] and neural networks by Stoianov and Nerbonne [81] and Stoianov [80, ch. 4]. The work presented here was originally presented in the Student Session of ESSLLI 2001 [34] and subsequently published as special issue of WEB-SLS [35].

This chapter describes experiments on the task of constructing from examples a model of Dutch monosyllabic words. The reason for restricting the domain is to avoid the added complexity of handling syllable boundary phonological processes. Furthermore by not using polysyllables no prior commitment is made to any one particular syllabification (and thus syllable structure) theory.

5.1 Extracting the Data

As a starting point, a rough template matching all syllables is assumed. If $\mathcal{C}$ is the set of all consonant segments and $\mathcal{V}$ the set of all vowels and diphthongs that appear in Dutch, this template is C_3VC_5, where $C_n \in \mathcal{C}^n \cup \ldots \cup \mathcal{C} \cup \oslash$ represents any consonant cluster of length up to n and $V \in \mathcal{V}$ any vowel or diphthong. The problem can now be reformulated as two single-predicate learning tasks where the target theory is one of acceptable affixes to a given

vowel and partial consonant cluster. The rules for prevocalic and postvocalic affixing are induced in two separate learning sessions.

The training data is derived from the 5692 monosyllabic words found in the Dutch section of the CELEX Lexical Database [59]. The positive examples are constructed by breaking the phonetic transcriptions down to three parts: a prevocalic and a postvocalic consonant cluster (consisting of zero or more consonants) and a vowel or diphthong. The consonant clusters are treated as 'affixes' to the vowel, so that syllables are constructed by repeatedly affixing consonants, if the *context* (the vowel and the pre- or postvocalic material that has been already affixed) allows it. So, for example, from the word /ma:kt/ (Dutch 'maakt') the following positives would be generated:

```
prefix(m,[],[a,:]).    suffix(k,[],[:,a]).
prefix(^,[m],[a,:]).   suffix(t,[k],[:,a]).
                       suffix(^,[tk],[:,a]).
```

where the context lists in suffix rules is reversed, so that the two processes are exactly symmetrical and can use the same background predicates.

What needs to be noted at this point is the representation of long vowels and diphthongs as lists of two symbols, instead of having unique symbols for each possible syllable nucleus. This should not be read as being phonologically or linguistically motivated, but rather as a computational convenience, since this representation allows for concepts like 'long vowel', 'diphthong' or 'short vowel' to be accessible by the ILP algorithm through the generic list manipulation predicates (`head/2` and `rest/2`, see also section 5.2 below) alleviating the need to bloat the background with explicit vowel length classification predicates.

The caret, `^`, is used to mark the beginning and end of a word. The reason that the affix termination needs to be explicitly licensed is so that it is not assumed by the experiment's setup that all partial sub-affixes of a valid affix are necessarily valid as well.

In Dutch, for example, a monosyllable with a short vowel has to be closed, which means that the null suffix is not valid. The end-of-word mark allows for this to be expressible as a theory that does not have the following clause: `suffix(^,[],[V]).`

The positives are, then, all the prefixes and suffixes that must be allowed in context, so that all the monosyllables in the training data can be constructed: 11067 and 10969 instances of 1428 and 1653 unique examples, respectively.

Negative data is constructed from randomly generated words that match the C_3VC_5 template and do not appear in the corpus. These are considered to

be non-words that should be accepted by the constructed theory. The random generator is biased so that the number of examples at each affix length is balanced, in order to avoid having the large numbers of long, uninteresting sequences overwhelm the shorter, more interesting ones.

To illustrate this last point, consider words where, for example, /v/ is the inner-most prefix to a vowel V: at this point there are some single consonants that may be prefixed (as in, for example, /kvark/, 'kwark') but no two consonants. This makes negative examples like `prefix(r,[v],[V])` and `prefix(f,[v],[V])` more useful that examples like `prefix(k,[r,v],[V])` and `prefix(l,[r,v],[V])`, because the former are helping identify a more complex and difficult to learn boundary. There are, however, $|\mathcal{C}| = 22$ possible ways to prefix one consonant but $|\mathcal{C}|^2 = 484$ ways to prefix two, meaning that in a uniformly selected sample there will be 22 times as many examples from the latter (less interesting) locution than from the former one.

These non-words are then split into two parts: one will be used for deriving the negative data and the other for evaluation. The negative examples are derived by the following deductive algorithm:

1. For each example, find the maximal substring that is provable by the positive `prefix/3` and `suffix/3` clauses in training data. So, for example, for /mtratk/ it would be `trat` and for /mlat/, `lat^`.

2. Choose the clause that should be a negative example, so that this word is not accepted by the target theory. Pick the inner-most one on each side, i.e. the one immediately applicable to the maximal substring computed above. For /mlat/ that would be `suffix(m,[l],[a])`. /mtratk/, however, could be negative because either `prefix(m,[tr],[a])` or `suffix(k,[t],[a])` are unacceptable affixations. In such cases, pick one at random. This is bound to introduce false negatives, but no alternative could be devised that does not presuppose at least part of the solution.

3. Iterate, until enough negative examples have been generated to disprove all the words in the negative training data.

The underlying assumption is that the space of monosyllables is very 'dense' or 'saturated' in the sense that almost all of the monosyllables allowed by Dutch phonology and syllable structure are actual words and the randomly generated words are much more likely to be missing from the corpus because they fall into a systematic gap rather than due to an accidental gap.

The simplest way to generate negative examples would have been to generate random prefix and suffix clauses that do not appear as positive examples. It would, however, *still* be necessary to generate non-words for

the purposes of evaluating the accuracy of the theory at the word level (as opposed to the level of each application of the affix predicates). For this reason it was deemed more consistent to generate negatives only once, for both training at the individual affixation level and the final evaluation at the word level.

5.2 The Background Knowledge

The *background knowledge* plays, as seen in section 2, a decisive role in the quality of the constructed theory, by implementing the theoretic framework to which the search for a solution will be confined. In more concrete terms, the background predicates are the building blocks that will be used for the construction of the hypothesis' clauses and they must be defining all the relations necessary to discover an interesting hypothesis.

Since the problem is, in effect, that of identifying the sets of consonants that may be prefixed or suffixed to a partially constructed monosyllable, the clauses of the target predicate must have a means of referring to various subsets of $\mathcal{C}$ and $\mathcal{V}$ in a meaningful and intuitive way. This is achieved by defining a (possibly hierarchical,) linguistically motivated partitioning of $\mathcal{C}$ and $\mathcal{V}$. Each partition can then be referred to as a feature-value pair, for example LAB+ to denote the set of the labials or VOIC+ for the set of voiced consonants. Intersections of these basic sets can then be easily referred to by feature-value vectors; the intersection, for example, of the labials and the voiced consonants (i.e. the voiced labials) is the feature-value vector [VOIC+,LAB+].

For the purposes of this task, they have been defined as relations between individual phones and feature values, e.g. `labial(m,+)` or `voiced(m,+)`. Feature-value vectors can then be expressed as conjunctions like, for example, `labial(C,+)` $\wedge$ `voiced(C,+)` to mean the voiced labials.

Except for the linguistic features predicates, the background knowledge also contained the `head/2` and `rest/2` list access predicates. These are the standard Prolog library predicates that match a list against its first element and against its tail (everything but the head), respectively. The second element `E` of a list `L` would then by accessed by `rest(L,L1),head(L1,E)` and so on. This approach was chosen over direct list access with the `nth/3` predicate, as bias towards rules with more local context dependencies.

Special note should be made here to the nucleus, where (as noted in the previous section) long vowels and diphthongs are represented as lists of length 2 (e.g. `[a,:]` or `[ɑ,u]`) instead of having a separate symbol for each vowel, long vowel and diphthong. This means that concepts like 'is a simple,

long vowel' or 'is a diphthong or long vowel' can be expressed through `head/2` and `rest/2` without having to introduce further background predicates that capture these relations:

```
rest(N, :).                % N is simple, long vowel
rest(N, N1), rest(N1, _). % N is diphthong or long vowel
```

The background knowledge described in sections 5.4, 5.5 and 5.6 below, encodes increasingly more information about Dutch phonology as well as Dutch phonotactics: for the experiment in 5.4 the learner has access to the way the various symbols are arranged in the IPA table, whereas for the experiment in 5.5 a classification that is sensitive to Dutch phonological processes was chosen. And, finally, in section 5.6 the *sonority level* feature is implemented, which has been proposed with the explicit purpose of solving the problem of Dutch syllable structure.

The quantitative evaluation given in the following four sections was done by 10-fold cross-validation. The 5692 monosyllables were randomly split in 10 sections and the example generation-training-evaluation cycle was repeated 10 times with a different section reserved for testing at each iteration. The recall and precision figures reported are the mean (and standard deviation) of the 10 recall and precision figures calculated from the 10 different datasets.

5.3 The Baseline theory

The training examples themselves can be regarded as a theory consisting of fully instantiated clauses only. When 10-fold cross-validated, the 'theory' had a mean recall of 47.07% ($\sigma^2 = 2.23$) with 98.88% ($\sigma^2 =$ 91E-4) mean precision. This recall will be considered the performance baseline for the recall of subsequent experiments. The precision is given only for completeness' sake, but it cannot be considered a baseline in any sense, since such an over-specific theory is expected to reject almost all negatives.

5.4 The IPA Chart

The International Phonetic Association's International Phonetic Alphabet [2] is collecting all possible phones in a chart organised per place and manner of articulation. Furthermore, each position in the chart hold two phones; consonants can be voiced or not and vowels can be rounded or not.

The organisation of the IPA chart can be seen as two, disjoint, spaces, one of consonants and one of vowels, with a feature vector for each phone

	Bilabial	Labio-dental	Alveolar	Post-alveolar	Palatal	Velar	Glottal
Plosive	p b		t d			k g	
Nasal	m		n			ŋ	
Trill			r				
Fricat.		f v	s z	ʃ ʒ		χ ɣ	h
Approx.		ʋ			j		
Lat.Appr.			l				

Table 5.1: IPA Chart, Consonants. For each position, the voiced consonant is on the right.

	Front	Central	Back
Close	i,ɪ y	ʉ	u
Close-mid	e ø		o
Mid		ə	
Open-mid	ɛ œ		ʌ
Open	a		ɑ ɒ

Table 5.2: IPA Chart, Vowels. For each position, the rounded consonant is on the right./i/ and /ɪ/ can only be distinguished by explicitly referring to one or the other.

specifying the position in this space. This provides us with a way to break down the phonetic inventory of Dutch into various subsets depending on its purely phonetic properties and without taking into account any peculiarities of Dutch phonology. Tables 5.1 and 5.2 show the parts of the IPA consonant and vowel tables that are pertinent to Dutch.

5.4.1 Design

The background predicates for this experiment are placing each phone in its position on the IPA chart along the dimension (feature) that each predicate is encoding. With the relevant parts of the IPA chart reproduced in tables 5.1 and 5.2, it can be seen that five such predicates must be defined: `place/2` and `manner/2` referring to both consonants and vowels, `voiced/2` for consonants, and `rounded/2` for vowels.

These predicates are extensionally defined to relate each value of the feature they implement with all the phones that carry that value. So, for example, some of the clauses of the `manner/2` predicate are:

```
manner(plosive, p). manner(plosive, b).
manner(nasal, m).   manner(nasal, n).
manner(open, a).    manner(open, ɑ).    manner(open, ɒ).
```

and so on.

Note that the end-of-word mark has no phonological features whatsoever and it does not belong to any of the partitions of either $\mathcal{C}$ or $\mathcal{V}$.

5.4.2 Results

The evaluation function used was the Laplace estimated accuracy (see Section 2.2.3). Since the randomly generated negatives must also contain false negatives, it cannot be expected that even a good theory will fit it perfectly. In order to avoid over-fitting, the learning algorithm was set to only require an accuracy of 85% over the training data.

The resulting hypothesis consisted of 199 prefix and 147 suffix clauses and achieved a recall rate of 99.3% with 89.4% precision.

All the false negatives were rejected because they couldn't get their onset licensed, typically because it only appears in a handful of loan words. The /ʤ/ onset necessary to accept 'jeep' and 'junk', for example, was not permitted and so these two words were rejected.

The most generic rules found were:

```
prefix(A,B,C) :- A= '^'.
prefix(A,[],C).

affix(A,B,C) :-  A= '^'.
affix(A,[],C).
```

meaning that (a) the inner-most consonant can be anything, and (b) all sub-prefixes (-suffixes) of a valid prefix (suffix) are also valid.

There is also a few pairs of rules that only differ in a couple of literals. This suggests that if a richer feature system would include features that effectively disjoin the features of this system, these pairs could be collapsed to one rule each. To give an example, consider these two rules:

```
prefix(A,B,C) :-
   head(B,D), manner(approx,D), head(C,E), length(short,E),
   place(front,E), voiced(minus,A).
prefix(A,B,C) :-
   head(B,D), manner(approx,D), head(C,E), length(short,E),
   place(front,E), manner(plosive,A), place(alveolar,A).
```

The first rule prefixes devoiced consonants to <approximant><short front vowel> sequences, for example allowing the /k/ in /kwɪlt/ 'quilt'. The second rule prefixes alveolar plosives to the same sequences, allowing words like /dwɪŋ/ 'dwing'. Devoiced alveolar plosives in words like /twɪst/ 'twist' are licensed by both rules.

These two rules could have been collapsed to one if a feature like 'devoiced consonant or alveolar plosive' was available. This particular disjunction might be unintuitive or impossible to independently motivate, but it suggests that a redundant feature set might allow for more compact theories than the minimal, orthogonal one used for this experiment. This is particularly true for a system like Aleph, that performs no predicate invention or background theory revision.

5.5 Feature Classes

For this experiment a richer (but more language-specific) background knowledge was made available to the inductive algorithm, by implementing the feature hierarchy suggested by Booij [4, ch. 2].

5.5.1 Design

The various phones are, again, accessed by means of feature vectors, but in this case the features are not orthogonal dimensions, but are forming the feature hierarchy shown in figure 5.1.

In this figure features are given in [brackets], with the remaining symbols being *feature classes*. All features are binary, and not all segments carry all features. Bearing a feature, however, makes it obligatory to also bear all its ancestors all the way to the root.

It follows that the *major class features* of the root node are always present. These are the features CONSONANT and SONORANT that divide the segment space into vowels [CONSONANT−, SONORANT+], obstruents [CONSONANT+, SONORANT−] and sonorant consonants [CONSONANT+, SONORANT+]. Since all vowels are sonorous, [CONSONANT−, SONORANT−] is an invalid combination, a restriction which is encoded in the background knowledge.

The features specifying the continuants, nasals and the lateral /l/ are positioned directly under the root node, with the rest of the features bundled together under two *feature classes,* those of the *laryngeal* and the *place* features.

Features that are bundled together in feature classes have to be all present or all missing from the specification of a segment, and their presence also

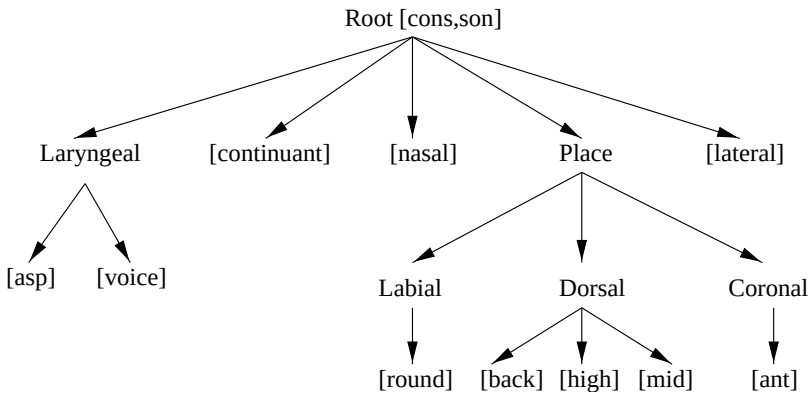


Figure 5.1: The feature geometry of Dutch

marks the segment as belonging to a certain class. So, for example, all (and only) laryngeals bear the voiced-voiceless distinction, with ASPIRATION separating /h/ from the rest. Feature classes are chosen so that they collect together features that behave as a unit in phonological processes of Dutch. In other words, the laryngeal features are bundled together because (a) it is useful to be able to collectively refer to them when talking about Dutch phonology, and (b) there are some segments for which all of these features are pertinent, but no segments for which only some of these features are pertinent.

It has to be noted that when it was mentioned that all features of a class have to be either present or missing, we had in mind features appearing in a segment's definition. When employing the feature hierarchy to refer to sets of features, under-specified feature vectors such as [CONSONANT+, BACK+] are both meaningful and useful.

Furthermore some derived or redundant features such as GLIDE, APPROXIMANT and LIQUID are defined. The vowels do not include the schwa, which is set apart and only specified as SCHWA+.

5.5.2 Results

The Prolog implementation of model described above consists of a database of segments where the values for all the features of each segment are stored. For example:

```
seg(k ,+, -, larynx(-, -), -, -, -, [dorsal(+,+,-)] ).
```

specifies /k/ as being [CONSONANT+, SONORANT−, larynx(VOICED−, ASPIRATED−), CONTINUANT−, NASAL−, LATERAL−, Place(dorsal(BACK+, HIGH+, MID−))].

In addition feature-access methods like the following are provided:

```
sonorant(Seg, Son):-  seg(Seg, _, Son, _, _, _, _, _).
larynx(Seg, Lar)  :-  seg(Seg, _, _, Lar, _, _, _, _).

   voiced(larynx(Val,_), Val).
aspirated(larynx(_,Val), Val).
```

to access the data in the `seg/8` terms. The relevant semantic information is also provided in the background, so that the variables used as input in `voiced/2` and `aspirated/2`, will always be `larynx/2` terms:

```
:- mode(1, larynx(+seg,-lar) ).
:- mode(1, voiced(+lar,-voic)).
```

The fact that the, say, larynx-related features are placed together under one term, means that they can be referred to individually but also collectively; whereas, for example, CONTINUANT and NASAL cannot. So it is possible to express the concept 'same laryngeal features' with fewer body literals that the concept 'same CONTINUANT and NASAL features':

```
same_laryngeal(Seg1, Seg2) :-
   larynx(Seg1, L), larynx(Seg2, L).
same_cont_nasal(Seg1, Seg2):-
   cont(Seg1, C), cont(Seg2, C),
   nasal(Seg1,N), nasal(Seg2,N).
```

which constitutes bias towards trying first the relation which the phonological model is considering as more common in Dutch phonology, or somehow more interesting for Dutch phonological descriptions.

Using the Laplace estimated accuracy as evaluation function and the background described above, the constructed theory consisted of 13 prefix and 93 suffix rules, accepting 94.2% of the test positives and under 7.4% of the test negatives.

Among the rejected positives are loan words ('jeep' and 'junk' once again), but also all the words starting with perfectly Dutch /s/ - obstruent - liquid clusters like the onset of /stra:t/ 'straat'. The prefix rule with the widest coverage is:

```
prefix(A,B,C) :-
   head(C,D), sonorant(D,plu), rest(B,[]).
```

or, in other words, 'prefix anything before a single consonant before a nucleus other than the schwa.'

The suffix rules were less strict, with only 3 rejected positives, 'branche', 'dumps' and 'krimpst' that failed to suffix /ʃ/, /s/ and /s/ respectively. Note that the first two of the false negatives are loan words and the remaining one is a superlative. Dutch superlatives are notoriously difficult to handle, and hand-crafted models of Dutch phonotactics also have to make special rules to treat them.

Some rules achieve wide coverage (although never quite as wide as that of the prefix rules,) but some make reference to individual phonemes and are of more restricted application. For example:

```
suffix(A,B,C) :-
   rest(C,D), head(D,E), rest(B,[]), A=t.
```

or, 'suffix a /t/ after exactly one consonant, if the nucleus is a long vowel or a diphthong.'

Of some interest are also the end-of-word marking rules (see in section 5.1 above about the ^ mark), because of the fact that open, short monosyllables are very rare in Dutch (there are four in CELEX: 'schwa', 'ba', 'hè', and 'joh'). This would suggest that the best way to treat those is as exceptions, and have the general rule disallow open, short monosyllables. What was learnt instead was a whole set of 29 rules for suffixing ^, the most general of which is:

```
postvoc(A,B,C) :-
   head(B,t), larynx(t,E), rest(B,F),
   head(F,G), larynx(G,E), A= '^'.
```

or 'suffix an end-of-word mark after at least two consonants, if the outer-most one is a /t/ and has the same values for all the features in the Laryngeal feature class as the consonant immediately preceding it.' In fact, suffix suffix obstruent (non-sonorant consonant) clusters in Dutch are always devoiced, and thus share their laryngeal features. This rule displays this phenomenon, although fails to capture it in its complete generality.

A final note that needs to be made regarding this experiment, is one regarding its computational complexity. Overlapping and redundant features might offer the opportunity for more interesting hypotheses, but are also making the search space bigger. The reason is that overlapping features are diminishing the effectiveness of the inverse resolution operator at keeping uninteresting predicates out of the bottom clause: the more background predicates can be used to prove the positive example on which the bottom clause is seeded, the longer the latter will get.

phoneme	obstruents	m	n	l	r	glides	vowels
sonority	1	2	2.25	2.5	2.75	3	4

Table 5.3: The Sonority Scale

5.6 Sonority Scale

The baseline theory of section 5.3 serves as a lower limit for the machine-learnt theories, in terms of size as well as predictive power: it simply stores the examples (thus performing no compression) and makes no generalization from the original data (thus performing very poorly on unseen data). To better appreciate the performance and compression rates of the two experiments described above, a hand-crafted model of Dutch syllabic structure is implemented as an upper limit of how well it is possible for any theory to perform. The model implemented is the one suggested by van der Hulst [87, ch. 3].

5.6.1 Design

The Dutch syllable is analysed by van der Hulst as having three prevocalic and 5 postvocalic positions, (some of which may be empty) and constraints are placed on the set of consonants that can occupy each.

The most prominent constraint is the one stipulating a high-to-low *sonority* progression from the nucleus outwards. Each phoneme is assigned a sonority value (table 5.3) based not only on language-independent features such as it being a SONORANT or an OBSTRUENT, but also because of syllable structure of Dutch itself. Especially the fine tuning done with respect to the sonority values of the nasals and the liquids is explicitly justified by the need to filter out impossible consonant clusters that would otherwise be predicted by the simpler model. It must, therefore, be noted that the one of the 'background predicates' is used is not only language-specific, but is also directly aimed at solving the very problem that is being investigated.

In addition to the high-to-low sonority-level progression from the nucleus outwards, there are both *filters* and explicit licensing rules. The former are restrictions referring to sonority (e.g. 'the sonority of the three left-most positions must be smaller than 4') or other phonological features (e.g. the 'no voiced obstruents in coda' filter in p. 92) and are applicable in conjunction with the sonority rule. The latter are typically restricted in scope rules that take precedence over the sonority-related constraints mentioned so far. The left-most position, for example, may be /s/ or empty, regardless of the

contents of the rest of the onset.

5.6.2 Implementation and Results

The core clauses in the implementation of the model were the ones enforcing the sonority-level progression:

```
suffix(A, [B|_], _) :-
        sonority(A, SA), sonority(B, SB), SA < SB.
```

The additional filters required by the model are implemented as predicates like the following one:

```
filter1(A) :- larynx(A, Lar), voiced(Lar, -).
filter1(A) :- sonorant(A, +).
filter1(^).
```

which directly implements the 'no voiced obstruents in coda' filter. As it can seen from the example, whenever it was necessary to refer to phone classes, the 'background predicates' used were the ones from Booij's feature-classes model of Dutch phonology (see section 5.5).

These filter predicates are then added to the body of the licensing rules to enforce the restrictions, so that the `suffix/3` clause given above would actually be:

```
suffix(A ,[B|_], _) :- filter1(A),
        sonority(A,SA), sonority(B,SB), SA < SB.
```

In total, the implementation required 11 clauses (three prefix and eight suffix clauses and filter-predicate clauses), but it must be noted again that one of the predicates referred to (the `sonority/2` relation) is extremely informed with respect to the problem at hand, so this lower size limit is not tight if one restricts the background predicates to not be informed about the actual solution of the problem.

This basic sonority progression rule as well as the most widely-applicable filters and rules[1] yielded an impressive compression rate and it also performed comparably to the two previous experiments, at 93.1% recall and 83.2% precision.

[1]Some were left out because they were too lengthy when translated from their fixed-position framework to the affix licensing one used here, and were very specifically fine tuning the theory to individual onsets or codas.

5.7 The Search Space

As mentioned in section 5.2 above, the background predicates should be providing meaningful and intuitive ways of accessing subsets of $\mathcal{C}$ and $\mathcal{V}$. One important aspect of the background theory is how finely grained it is and how many different ways or overlapping viewpoints it is providing for accessing these subsets. The most useful background theories will then be the ones that are most informed about the target concept; that is, the ones that most tightly circumscribe the minimum background necessary to express the target concept: at the limit all and only those background predicates will be present that are to be found in the bodies of clauses of the (ideal) target predicate.

The reason for this, as explained below, is that richer background theories might allow for more interesting hypotheses to be constructed, but they also imply an increase in the size of the bottom clause and, subsequently, the space within which the search for a hypothesis has to be performed.

5.7.1 Bottom Clause Size

In the experiments described in sections 5.4 and 5.5 the 'grain' of the background predicates was the same (that is, all individual phones were addressable), but the feature-class background of sections 5.5 was, nevertheless, richer and allowed more ways to access subsets of $\mathcal{C}$.

The bottom clause is a minimal generalization of an example, where all ground atoms have been replaced by variables, and a body has been constructed where these variables appear in the body literals in all the ways that are (a) permitted by the semantics defined for the predicates that appear as body literals, and (b) consistent with the requirement that the original example and only that can be derived from the bottom clause. (See section 2.3.2 about the bottom clause in general and 2.4 about the specifics of the Progol algorithm used here.)

In more concrete terms, this means that in the context of a richer, more redundant background theory the saturation (that is, bottom-clause construction) process will yield longer bottom clauses. This is corroborated by experimental data taken from the two experiments of this chapter: when saturating all the available positive examples, the mean bottom clause length is 14.38 literals ($\sigma^2 = 2.58$) for the IPA experiment and 26.35 ($\sigma^2 = 7.95$) for the Feature Classes experiment. The density functions of the bottom clause sizes can be seen in Fig. 5.2, estimated on a Gaussian kernel (bandwidth = 4). What can be seen in that figure in particular, is that the high deviation of the second experiment's bottom clause size is due to the bi-modality

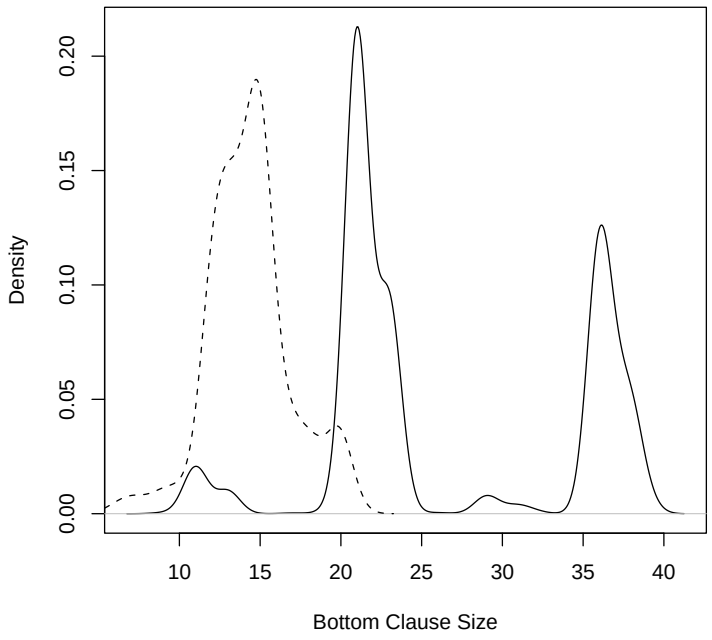


Figure 5.2: The density function of the Bottom Clause sizes. The dashed line is from the IPA experiment (section 5.4) and the solid line from the Feature Geometry experiment (section 5.5).

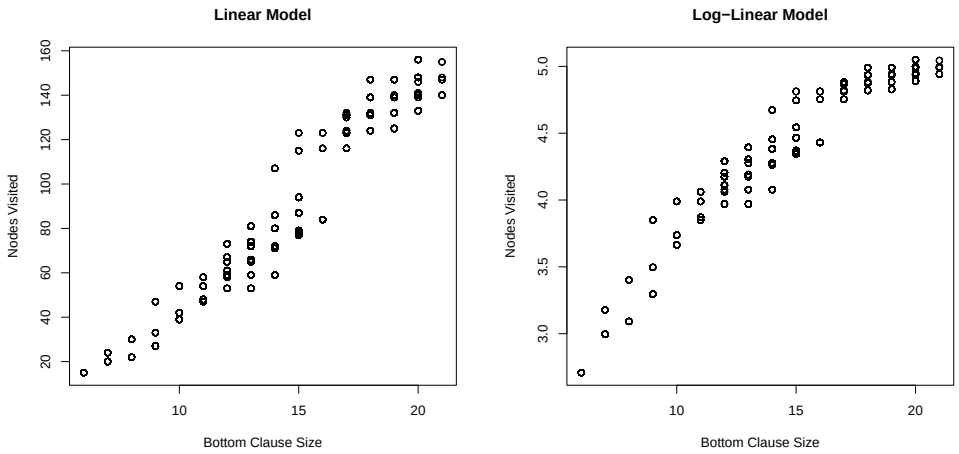


Figure 5.3: The growth of the reduction effort against the bottom clause ength. Their relation appears to be sub-exponential.

that it exhibits, with most of the bottom clauses clustering around length 22 and length 36, whereas the the curve derived from IPA experiment is much smoother.

5.7.2 Search-Space Size

The advantage of not including background predicates that will not be employed by the final theory is that it reduces the space that needs to be searched, since its size is directly related to the number of background predicates that are present in the bottom clause constructed for each example that gets saturated. In the absence of syntactic bias to restrict the ways in which background predicates can be combined, all possible subsets of the bottom clause are valid generalizations of the example and the search space consists of the power-set of the body of bottom clause; it thus grows exponentially as a function of the size of the bottom clause.

In practice, the number of nodes actually visited is smaller that the total size of the search space. Fig. 5.3, for example, plots the number of nodes actually visited during each reduction vs. the size of the bottom clause that seeded that particular reduction (data is taken from the IPA experiment). Although quite sharp, the log-linear curve seems to be sub-exponential.

This sharp increase in the size of the search can be checked by imposing very restrictive syntactic bias, so that the actual search space is much smaller

than the raw space defined by the bottom clause. It can also be by-passed by employing a high-quality, informed evaluation function that (acting as a heuristic) will quickly guide the search towards a satisfactory clause without having to consider most of the space. Or, finally, the space can be kept small enough to be searched efficiently by keeping the size of the bottom clause small. It can be seen then that in all three cases the key is prior knowledge available regarding syntactic and semantic properties of the target concept itself, as contrasted to prior knowledge in general which includes information about the target as well as the 'universe' in which it operates.

In most non-trivial applications this information will be limited, and its discovery the very task of ILP. The usual trade-off is then between the computationally expensive choice of including all relations known to hold in the experiment's universe and imposing no syntactic restrictions on the one hand, and on the other hand severely restricting the search space and running the risk of missing good solutions. This trade-off can also be seen when comparing the background described in section 5.4 with that of section 5.5 above. Although the grain is equally fine in both experiments (in the sense that it is possible to refer to all possible subsets of consonants and vowels), the former experiment (based on the IPA table) employs a system of orthogonal features that implement a minimal (in terms of features $\times$ values) set of methods to access the members of $\mathcal{C}$ and $\mathcal{V}$. By contrast the feature hierarchy described in section 5.5 is richer in that it offers multiple ways of looking at the data, which amounts to being redundant. This allows for potentially more interesting theories to arise, but also results in longer bottom clauses and a larger search space.

5.7.3 Data-Parallelism Vs. Or-Parallelism

Predicates consist of multiple clauses which represent multiple ways for the predicate to be satisfied; individual clauses might also include explicit OR operators in their bodies; finally there might be multiple ways to instantiate the variables in a clause's literals. All these situations constitute *choicepoints* for a Prolog engine, where one of many alternative paths through the proof search space has to taken. The Prolog backtracking mechanism will exhaust each path before backtracking to the last choicepoint and trying the next option, and so on until the goal is satisfied or there are no choices left and the goal fails.

An Or-parallel Prolog implementation is one that tries all the clauses at a choicepoint in parallel. Or-parallelism can be used to divide the search space of the ILP algorithm in 'sectors' which will be searched in parallel by the nodes of the machine, since the ILP algorithm's search is actually the search

for a proof: each time a literal is added to the clause under construction it is picked from the 'pool' of literals provided by the bottom clause. In (very schematic) Prolog this is implemented along the lines of this code fragment:

```
add_one_literal(C, NextC):-
   bottom_clause( (Head:-Body) ),
   member(Lit, Body),
   append(Lit, C, NextC),
   is_good_clause(NextC).
```

which will backtrack back to `member/2` each time `is_good_clause/1` is not satisfied, until all the member of the bottom clause have been tried out. An Or-parallel Prolog implementation would employ the nodes of a parallel machine to try out all the ways to instantiate `Lit` in `member(Lit,Body)` in parallel.

The data-parallelism described in chapter 3, by comparison, is a parallel implementation of the evaluation function employed by `is_good_clause/1` to decide whether to accept the current clause or not. It should, then, be noted that the computation expense discussed here cannot be treated by data-parallelism, since most of the time is consumed in constructing candidate clauses and traversing the search space, which means that the bottleneck is not the large amount of data against which each hypothesis needs to be tested.

5.8 Conclusions

The quantitative results from the ILP experiments presented above are collected in Table 5.4, together with those of an abductive approach to the same problem by Tjong Kim Sang and Nerbonne [83, Section 4.3], and the results from the baseline and the sonority scale implementations. Those last ones in particular are listed for comparison's sake and as the logical end-point of the progression towards more language- and task-specific prior assumptions.

The second and third rows are directly comparable, because they both refer only to phonetic primitives without any phonologically motivated background knowledge. Furthermore the fact that the C_3VC_5 template assumed in this work is not taken for granted in the abductive experiment is taken into account in terms of compactness as well as performance, since (a) the 1154 rules of Table 5.4 only refer to affixation to an already formed 'basic word', which is described by 41 extra rules not included in the affix-rule count, and (b) in the abductive experiment precision is measured on random strings, whereas here only strings matching the C_3VC_5 template are

	Recall	Precision	Size	
Baseline	47.1%	98.9%	12133	13171
(Tjong et al., 2000)	99.1%	74.8%	577	577
(Stoianov, 2001), $\theta = 1.6$	95.0%	95.0%		
(Stoianov, 2001), $\theta = 1$	99.0%	81.0%		
IPA	99.3%	79.8%	145	36
Feat. Classes	94.2%	92.6%	13	93
Sonority	93.1%	83.2%	3	8

Table 5.4: Results

used. As can be seen, then, the ILP-constructed rules compare favourably (in both performance and hypothesis compactness) with those constructed by abduction.

One should also make a note of the results reported by Stoianov and Nerbonne [81] and Stoianov [80]. There, neural networks (and, more precisely, SRN) were trained to predict the following phoneme in a word based on the candidate word so far. When using the network on unseen data, Stoianov [80, p. 90] plots the error rates for different thresholds[2] where the negative data is randomly generated strings fitting a C_4VC_3 template (idem., p. 83).

If we pick the optimal threshold to be the point where the false negatives and the false positives error rates are equal, that is, the intersection of the lines labelled (A) and (I) on the graph (threshold $\theta = 1.6$), we get an error rate of 5%. If we assume equal number of positive and negative examples, then we can interpret the error rates in the graph as precision and recall of 95%, slightly better than the results of the Feature Classes experiment. If we pick the point $\theta = 1$ where the false negatives are comparable to the 99.3% recall of the IPA experiment, the false positives approach 22%, yielding a precision of around 81%. (always under the assumption that the number of positive and negative examples is equal.) Please note the assumptions made in this paragraph and also that these calculations are rather rough, since the figures used are read off a graph and are not the exact measurements. They do, however, suggest similar performance as the one achieved here.

What can be also seen by comparing the two ILP results with each other, is that the drop in recall between the the third and fourth row is compensated by higher precision and compression, suggesting a direct correspondence between the quality of the prior knowledge encoded in the background theory and that of the constructed hypothesis.

[2]The threshold is effectively the point of balance between favouring recall and favouring precision.

One interesting follow-up to these experiments would be attempting to expand their domain to that of syllables of multisyllabic words and, eventually, full word-forms. In the interest of keeping the problems of syllabic structure and syllable-boundary phonology apart, a way must be devised to derive from the positive data (i.e. a corpus of Dutch word-forms) examples for a distinct machine learning session for each task.

Given the discussion of section 5.7 on data-parallelism versus or-parallelism, it would also be interesting to port an ILP system such as Aleph to an or-parallel Prolog compiler. Such a system on which Aleph could be ported is YapOr [71], based on the Yap Prolog compiler.

Chapter 6

Conclusion

Exempting the first two chapters that introduce the reader to the history, theory, and implementation of ILP systems, this thesis has been concerned with the extension of an ILP system to allow for the evaluation of clauses in parallel (Chapter 3) and two applications of ILP; one on Syntax (Chapter 4) and one on Phonology (Chapter 5).

This concluding chapter will first summarise and discuss the computational cost issues encountered during these experiments (Section 6.1). Then the results and conclusions of the two experimental chapters will be brought together and summed up in Section 6.2. The points made there will be further explored in the discussion that follows in Section 6.3, where the experience of applying ILP to these linguistic tasks will be discussed and reflected upon.

The thesis closes by suggesting *future directions* for research in applying ILP for linguistic tasks in Section 6.4 — the inevitable admission that this work is not complete, since there are always more paths to be explored and ideas to be tried.

6.1 Computational Complexity

As already noted in the introduction of Chapter 3, ILP runs are notoriously slow and memory-intensive, which makes parallel computation a very attractive prospect. The attempt at a data-parallel version of Aleph described in that chapter did not, however, yield very promising results, demonstrating that the bottleneck is the size of the search space — that is, the number of clauses that need to be constructed and evaluated — rather than the cost of the evaluation phase per se.

What makes the problem even more acute is the observation made in Section 5.7 about the growth rate of the search space as a function of the size of

the bottom clause. Since the more complex, long and useful the background theory is, the longer a bottom clause it will produce, one can immediately see that interesting background theories will make data-parallelism even less attractive an option for ILP.

One thing that should be noted at this point is that the measurements for the data-parallel Aleph system were carried out on a workstation cluster rather than a parallel machine. Workstation clusters are arrays of workstations communicating via a network connection and not sharing memory, meaning that they suffer from a high communication overhead, by comparison to parallel machines. Furthermore, broadcasting — on which Aleph/MPI heavily relies — is especially affected by the fact that there is no shared memory from which all receiving nodes can copy the broadcast message, but a number of point-to-point connections has to be established instead.

On the other hand, clusters have to offer larger numbers of nodes, since a network is much more easily expendable than a parallel machine. This makes them more appropriate for highly parallelised tasks with little communication among nodes and near-linear benefits from each node added to the computation. Aleph/MPI on the other hand is not such an application: (a) there is regular communication between the nodes, which although not very voluminous does require that the overhead be repeatedly paid, and (b) Ahmdal's Law takes effect rather quickly, since a large part of the computation remains serial. This implies that Aleph/MPI would benefit more from a shared-memory multi-processor machine with low communication overhead and few nodes, than from a cluster with high communication costs and a much larger number of nodes.

6.2 Summation of Results

The evaluation of the ILP experiments conducted in Chapters 4 and 5 has been concerned with the quantitative as well as the qualitative analysis of the resulting theories. The quantitative results from the application of ILP to capture linguistic phenomena are encouraging, although not spectacular. In both cases very high compression is achieved: BaseNP chunking is, effectively, done with just 11 clauses (see Section 4.5) whereas the phonotactic rules of Dutch can be expressed with 181 or 106 clauses, depending on the informedness of the background theory (see Section 5.8).

But although ILP achieves high compression of the data, suggesting high generalization power, the precision and recall rates for the BaseNP chunking experiment are inferior to the competition: the ILP-induced chunker achieved a recall rate of 85.32% with 78.62% precision, which compares poorly with

System	ML Approach	Precision	Recall
Kudoh and Matsumoto	SVM (*)	93.45%	93.51%
Van Halteren	WPDV/MBL (*)	93.13%	93.51%
Tjong Kim Sang	MBL (*)	94.04%	91.00%
Zhou, Tey and Su	HMM	91.99%	92.25%
Déjean	ALLiS	91.87%	92.31%
Koeling	MaxEnt	92.08%	91.86%

Table 6.1: Performance of top chunkers from the CoNLL-2000 Shared Task. The starred (*) approaches reach a decision by combining the results of multiple learners of the type(s) shown on the table.

the top results from other approaches on the slightly more complicated task of BaseXP recognition, as can be seen from Table 6.1, the results of the CoNLL-2000 Shared Task [82]. (References to the papers describing each system also available ibidem.)

On the other hand, in the domain of Dutch phonotactics the ILP approach described in Chapter 5 shows considerably improved results by comparison to an earlier abductively induced theory [83] on both performance and compression (see Section 5.8). More specifically, the experiment with the simpler background (IPA table, Section 5.4) slightly improves upon the performance of the abductive theory with considerably fewer clauses. Furthermore, the more informed background (Booij's Feature Classes, Section 5.5) gives rise to a theory that is even more compact and considerably more precise.

The qualitative analysis of the resulting theories has also shown chunking (or at least the experimental setup chosen) to be ill-suited for applying ILP, since the promise of readability and conciseness of the resulting theory and the benefits from the usage of background knowledge — advertised as ILP's biggest advantages both in Section 2.6 and in the literature in general — are not fully substantiated. As already noted in the concluding section of the relevant chapter (Section 4.5), the word-tagging setup used distributes each chunking decision over a number of individual word-tagging decisions, which are to a large extend made independently. This makes it difficult to interpret the theory, since the chunks end up depending a lot on the interaction between rules. It also renders the background knowledge less useful, since some framework constraints (such as, for example, those imposed by X-bar theory) are not expressible in this setup.

A similar approach was chosen for the phonotactics experiments described in Chapter 5: the setup was such that the recognizer would make individual decisions about attaching a phoneme to an already recognized and accepted

'kernel.' In this case, however, this approach yielded much more interesting results: the resulting theories were more concise as well as more accurate than the abductively constructed ones, as shown in Section 5.8.

The promising result of the qualitative analysis of the chunking experiment is that what little background knowledge is available does get used in a meaningful way: a common pattern is clauses where the `naive/2` background predicate is used to assign tags to words, with the other literals in the clause restricting its domain of application. Similarly in the phonotactics experiments, the quality and sophistication of the background knowledge has an impact on the performance and size of the theory, also suggesting that the information encoded in the background does get used and does make a difference.

6.3 Discussion

It is more interesting to analyse the results of the ILP experiments in conjunction with the results of other systems on the same task, and a comparison of ILP with other machine learning approaches tackling the BaseXP chunking task was given above.

By examining the best performing systems (see Table 6.1 above), we find three systems combining the results of multiple chunkers to reach a decision at the top three positions: the Kudoh-Matsumoto system takes a majority-vote decision based the results of the complete set of pairwise classifiers; Weighted Probability Distribution Voting (WPDV) for Van Halteren; and majority voting for Tjong Kim Sang. Kudoh and Matsumoto later [38] improved the performance of their system even further, by adding another layer of voting and applying weighted-voting among 8 systems like the ones used for the CoNNL-2000 Shared Task.

Furthermore, the individual chunkers the decision is based on, are based themselves on numerical or stochastic Machine Learning algorithms[1] for the top two systems, and only at the third position appears a majority-voting combination of (symbolic) Memory-Based Learners. The single-chunker systems trail with Hidden Markov Models (Zhou, Tey and Su), followed by a symbolic theory-refinement system (ALLiS, employed by Déjean), and then the procession of non-symbolic approaches goes on with Maximum Entropy learners and Markov Models, and only at the very last places symbolic approaches appear again. Notice how all chunker combinations out-perform

[1]Support Vector Machines for Kudoh-Matsumoto and Weighted Probability Distribution Voting for Van Halteren, although the latter also includes one Memory-Based Learner among the five chunkers used.

all single-chunker systems, and within each of these two classes stochastic and numerical systems out-perform the symbolic ones, with the exception of ALLiS.

It seems, then, that systems that combine the results of multiple learners are better suited for the task, especially when they are combining the results of non-symbolic machine learning systems, and that ILP is simply the wrong tool for this kind of problem. Reflecting on the general properties of ILP, one notices that fuzzy and noisy concepts pose the biggest problems for the induction of logic programmes, and for formal logic in general. BaseNP chunking is such a noisy domain, since quite often the same string of part-of-speech tags will be chunked differently, depending on semantic or long-distance syntactic features that are outside the scope of the chunker. This is, naturally, not implying anything with regards to using logic formalisms in NLP in general, but only in tasks like chunking where the processing is performed on limited information so as to keep the processing fast and simple.

One can also see that the ILP learner suffered from the noise of the data in the large number of ungeneralized examples remaining at the end, versus the small number of general clauses successfully induced: eleven out of 160, as noted in Section 4.5. Relaxing the accuracy requirements would improve this ratio, but only at the expense of precision, yielding over-generalizing clauses.

Furthermore, as has already been noted, only a very limited amount of background knowledge was made available to the learning process, due to the very nature of the experiment's setup. This setup has become quite standard in the literature since the seminal work of Ramshaw and Marcus [69], but it seems to be ill-suited for the purposes of ILP. On the other hand, a more complex formalism — like the CFGs suggested by Abney [1] — would constitute an unjustified[2] overkill, since finite-state theories have been successfully induced for this problem. From the above we can conclude that chunking is a problem that is difficult in the wrong way, so that is does not benefit from the advantages of ILP but it does bring out its weaknesses.

And then again, a similar setup in the domain of phonotactics has yielded much better results. This can be attributed to the following factors:

- Phonotactic data is much less noisy: native Dutch words can be described consistently, with all the relevant features available to the learner. That is, no information[3] was excluded from the background, so all the elements of the decision are available. This leaves only relatively new loan-words that have not been phonologically dutchified as the sole

[2] computational complexity-wise

[3] As, for example, semantics was excluded for the chunking experiment.

source of noise, and those were appropriately treated as exceptional singularities.

- Phonology (unlike syntax) is an inherently local phenomenon, so that the prior knowledge and the theoretical framework is also more readily expressible in a local, finite-state formalism. In other words, no prior knowledge is wasted due to the difficulty or impossibility of expressing it within the experimental setup. The generalization power of ILP is, then, employed to compress the number of affix rules necessary to fully describe the phenomenon.

Schaffer [73] showed that a *conservation law* must hold for the generalization performance of each particular learning system. That is, it is not possible for a particular system to perform well on all domains, but performance increase in one dataset will have to be balanced by lower performance in some other datasets. In other words, the kind of generalizations a system is looking for will be appropriate for some problems, but not for others. This result is, of course, simply adding some extra theoretical support to well known maxims of Computer Science, such as *use the right tool for the job* because *there's no such thing as a free lunch!*

6.4 Future Directions

The discussion above provides some pointers towards interesting ways to follow up on the work described in this thesis, which should be read as complementing (rather than summarising) the partial future research suggestions at the end of each chapter.

Due to the highly disjunctive nature of predicate definitions in Logic Programming, ILP is well suited to capture the outlying data-points that remain after a numerical or stochastic system has explained the main body of the data. That is, of course, true of other symbolic systems as well. In fact, a combination system like Van Halteren's (see above) would rely on its Memory Based Learning component to do exactly that: to detect the exceptional cases that would either lie outside the Weighted Probability Distribution Voting theories or cause them to over-generalize.

It would, therefore, be interesting to see how well an ILP-generated chunker complements statistical chunkers in a combination system like the one of Van Halteren. This would provide data for a comparison between ILP and Memory Based Learning on the task of complementing a battery of stochastic systems.

In the light of the gains, in terms of results, achieved by improving the background theory in the phonotactics experiments, it would be interesting to experiment further with the linguistic knowledge in the chunking experiment as well. The linguistic background provided there was restricted to features like NOMINAL or VERBAL, which simply hinted at the syntactic properties of each part-of-speech, but provided no information other than various linguistically motivated ways to partition the set of part-of-speech tags. This proved much more effective in the phonotactics experiments, but — as already noted in the discussion above — phonotactics is an inherently local phenomenon and syntax is not.

What then emerges as an interesting possibility, is some experimentation which breaks with the long chunking-as-word-tagging, finite-state tradition that stems from the original Ramshaw and Marcus experiment. To be able to capitalize better on the strong points of ILP, the setup should be such that long-distance dependencies should be discouraged[4] but nevertheless explored, and kept if found to be making a distinction that cannot be otherwise made. Access to a dictionary (or, in any case, a more complex lexeme ontology than a simple part-of-speech tag) would also be something that would be easily incorporate-able as prior in ILP.

[4] In order to retain the fast-and-shallow character of chunking, and not gravitate towards a full parser.

Samenvatting

In de logica en de filosofie wordt onder de term inductie het proces verstaan waarbij natuurlijke wetten worden afgeleid door middel van het redeneren van onderdeel naar geheel, van het bijzondere geval naar het algemene of van het individuele naar het universele. Inductief redeneren voegt het totaal aan alle observaties of voorkomens samen tot een kortere wet die deze gebeurtenissen beschrijft en deductief kan voorspellen. Deductie is dan het redeneren vanuit tegengestelde richting: van de natuurlijke wet naar de observaties, van geheel naar onderdeel, van het algemene naar het bijzondere geval en van het universele naar het individuele.

Inductie tracht kennis te genereren, namelijk een hypothese die een verzameling empirische data of observaties verklaart binnen een kader van reeds aanwezige (achtergrond-)kennis. Inductief logisch programmeren is een machine learning principe dat inductief redeneren implementeert in het domein van logisch programmeren. Met andere woorden, gegeven een logisch programma B (achtergrondkennis) en D (data) proberen ILP algoritmes een logisch programma H (hypothese) te construeren zodat $B \wedge H \models D$.

In het typisch geval zal een ILP algoritme H opbouwen in stappen van één logische zin per iteratie met behulp van een incremental cover-strategie. Het zoeken naar de volgende logische zin wordt gedaan door een partiële ordening aan te brengen in de zoekruimte van alle mogelijke logische zinnen op een algemeen-specifiek-as. De operator die gebruikt wordt om deze ordening aan te brengen kan een generalisatie-operator zijn of een specialisatie-operator, wat respectievelijk resulteert in zoeken van specifiek naar algemeen of van algemeen naar specifiek. Het zoeken is aan de meest algemene kant gebonden door de lege, inconsistente zin $\square$, en aan de meest specifieke kant door de zogenaamde bottom clause, een minimale generalisatie van een positief voorbeeld. Progol is een dergelijk ILP algoritme en Aleph is een ILP systeem dat (onder meer) het Progol algoritme implementeert. Hoofdstuk 2 van dit proefschrift omvat een introductie in ILP in het algemeen en Progol en Aleph in het bijzonder.

Hoofdstuk 2 eindigt met een discussie over de keuze van ILP voor taalkun-

dige experimenten. ILP kent dezelfde voor- en nadelen als symbolisch rekenen in het algemeen: het vereist veel rekenkracht (zowel om een theory te construeren als om haar toe te passen) en het verwerken van numerieke data is lastig. Aan de andere kant, wanneer een symbolisch formalisme de voorkeur geniet, dan zijn logica en ILP de beste oplossing. Dat wil zeggen wanneer de uiteindelijke theorie niet alleen kwantitatief geëvalueerd en toegepast moet kunnen worden, maar ook de kwalitatieve analyse van de resultaten van belang is.

ILP is, zoals gezegd, een rekenkrachtintensieve taak, wat het een goed object voor parallellisatie maakt. Hoofdstuk 3 begint met een korte beschrijving van de *Message Passing Interface* (MPI). MPI is een specificatie van de *Application Programmers Interface* (API) voor message-passing libraries. Vervolgens wordt Aleph/MPI beschreven, een dataparallelle versie van Aleph die ontwikkeld is voor de uitvoering van het onderzoek dat in dit proefschrift beschreven wordt. Aleph/MPI is gebaseerd op een uitbreiding van het YAP Prolog systeem met een interface (eveneens ontwikkeld voor dit project) naar MPI libraries. Hoofdstuk 3 sluit af met het testen en evalueren van deze uitbreidingen en enige speculatie over de domeinen waarin zij toegepast zouden kunnen worden.

De volgende twee hoofdstukken behandelen de toepassing van ILP op twee taalkundige domeinen: shallow parsing en fonotactiek. In hoofdstuk 4 worden de experimenten met shallow parsing besproken en in het bijzonder een experiment waarin getracht werd door middel van inductie een base NP chunker voor het Engels af te leiden. Een dergelijke chunker herkent alleen NP's van het laagste niveau (BaseNP). De NP 'confidence in the pound' wordt bijvoorbeeld geanalyseerd als:

(1) [NP [N1 Confidence] [PP in [N1 the pound]]]

maar de NP-chunker analyseert deze NP simpelweg als:

(2) [Confidence] in [the pound]

De invoer voor de chunker is platte tekst, die normaliter geannoteerd is met part-of-speech-informatie. Chunking is een probleem dat eindige toestandenautomaten aankunnen in één enkele run over de geannoteerde tekst.

Het experiment in hoofdstuk 4 is zodanig opgezet, dat de chunker een syntactische tag toewijst aan ieder woord. Deze tag geeft aan of het woord wel of niet onderdeel is van een BaseNP. Het predikaat waarvoor een hypothese afgeleid moet worden is de relatie tussen een woord in een bepaalde context en een syntactische tag. De linker context bestaat uit woorden met een part-of-speech-label én een syntactisch tag, de rechter context bestaat uit

woorden die alleen geannoteerd zijn voor part-of-speech: de richting waarin over de tekst gegaan wordt is van links naar rechts. De geannoteerde tekst is afkomstig uit de Penn Treebank, een geannoteerd en geparseerd Engels corpus.

Kwantitatief kan de geconstrueerde theory de resultaten van stochastische leermethodes niet evenaren. Kwalitatief gesproken heeft theory de leesbaarheid en modulariteit van symbolische theorieën. Dit kan worden toegeschreven aan verschillende oorzaken, zoals de ruis die inherent is aan de data, maar ook de ruis die veroorzaakt wordt doordat de opzet van het experiment een lange afstandsverschijnsel zoals syntaxis in de vorm van een lokaal verschijnsel forceert.

In hoofdstuk 5 wordt vervolgens de toepassing van ILP in het domein van de fonologie besproken, en meer in het bijzonder de fonotactiek: de regels die bepalen welke opeenvolgingen van fonemen in een bepaalde taal zijn toegestaan en welke niet. Een fonotactisch model is met andere woorden een model dat de non-woorden van een taal in twee groepen indeelt: systematische gaten, die nooit woorden van de taal hadden kunnen zijn, en toevallige gaten, die woorden hadden kunnen zijn, maar het toevalligerwijs niet zijn.

De algemene opzet is vergelijkbaar met die van het chunkingexperiment: het doel is een Nederlandse woordherkenner die eenmaal vanuit de nucleus naar buiten toe over de lettergreep heengaat en beslist of het een mogelijke lettergreep is. Het experiment zoals dat hier beschreven wordt is beperkt tot eenlettergrepige woorden, ervan uitgaande dat de fonotactiek in zijn geheel bestaat uit verschillende problemen: het herkennen van mogelijke lettergrepen en het combineren van lettergrepen tot welgevormde woorden.

Het doel is een predikaat dat een klinker of tweeklank en een gedeelte van het pre- of postvokale materiaal verbindt met een verzameling fonemen waarmee het kan combineren tot een welgevormde lettergreep. Het zij daarbij opgemerkt dat het niet nodig is aan te nemen dat alle tussenliggende stadia van een welgevormde lettergreep eveneens welgevormd zijn. De data zijn afkomstig uit het Nederlandse gedeelte van de CELEX Lexical Database. Twee verschillende manieren om de Nederlandse medeklinkers weer te geven zijn geprobeerd, de een uitgebreider en gedetailleerder dan de andere. De resultaten worden met elkaar vergeleken en hierbij blijkt dat de ILP leermethode profijt heeft van de uitgebreidere achtergrondkennis, waardoor een theorie genereerd wordt die compacter is en bovendien beter resultaten levert.

Het laatste hoofdstuk bevat een bespreking en samenvatting van de conclusies die aan het eind van elk hoofdstuk getrokken konden worden, maar daarnaast worden in dit hoofdstuk algemene conclusies getrokken uit het project als geheel. In het bijzonder wordt gesuggereerd dat chunking (of

tenminste het experimentele kader dat hier gekozen is) niet geschikt is voor de ILP leermethode, omdat chunking het moeilijk maakt om voordeel te halen uit de aantrekkelijke eigenschappen van symbolisch machine learning, namelijk de mogelijkheid om expliciete achtergrondkennis te gebruiken en de leesbaarheid van de resultaten. Wat betreft de fonotactische experimenten wordt gesuggereerd dat deze meer succesvol waren omdat de fonologische data minder ruis bevatte dan de treebank en omdat fonologie een lokaal probleem is, wat het gemakkelijker maakt om het in het experimentele kader van de hoofdstukken 4 en 5 te passen zonder informatie te verliezen.

Περίληψη

Στὴν Λογικὴ καὶ τὴν Φιλοσοφία ὀνομάζουμε *ἐπαγωγὴ* τὴν διαδικασία ἀνακάλυψης φυσικῶν νόμων ἐργαζόμενοι ἀπὸ τὸ ἐπιμέρους πρὸς τὸ ὅλο, ἀπὸ τὸ εἰδικὸ πρὸς τὸ γενικό, ἀπὸ τὸ ἀτομικὸ πρὸς τὸ καθολικό. Ὁ ἐπαγωγικὸς λογισμὸς συμπυκνώνει τὸ σύνολο τῶν παρατηρήσεων ἢ ἐκφάνσεων ἑνὸς φαινομένου σὲ ἕναν φυσικὸ νόμο ποὺ τὶς περιγράφει συνοπτικὰ καὶ ἀπὸ τὸν ὁποῖο μποροῦμε *ἀπαγωγικὰ* νὰ λάβουμε τὶς ἀρχικὲς παρατηρήσεις. Ἀπαγωγὴ εἶναι, λοιπόν, ὁ λογισμὸς πρὸς τὴν ἀντίθετη κατεύθυνση: ἀπὸ τοὺς φυσικοὺς νόμους πρὸς τὶς παρατηρήσεις, ἀπὸ τὸ ὅλο πρὸς τὸ ἐπιμέρους, κ.ο.κ.

Ἡ ἐπαγωγη ἐπιχειρεῖ νὰ διατυπώσει μία *ὑπόθεση* ἡ ὁποία θὰ ἐξηγεῖ τὰ ἐμπειρικὰ δεδομένα (τὶς παρατηρήσεις) μέσα στὸ πλαίσιο τῆς *ἀπριορικῆς γνώσης* (prior knowledge) ποὺ λαμβάνουμε ὡς δεδομένη. Ὁ *Ἐπαγωγικὸς Λογικὸς Προγραμματισμός* (ΕΛΠ) εἶναι τὸ πεδίο τῆς Τεχνητῆς Νοημοσύνης (καὶ πιὸ συγκεκριμένα τῆς Μηχανικῆς Μάθησης) ποὺ ὑλοποιεῖ τὸν ἐπαγωγικὸ λογισμὸ στὸ πεδίο τοῦ Λογικοῦ Προγραμματισμοῦ. Μὲ ἄλλα λόγια, δοθέντος ἑνὸς λογικοῦ προγράμματος B (ἀπριορικὴ γνώση) καὶ ἑνὸς λογικοῦ προγράμματος D (δεδομένα), ἕνας ἀλγόριθμος ΕΛΠ ἐπιχειρεῖ νὰ κατασκευάσει ἕνα λογικὸ πρόγραμμα H (ὑπόθεση) τέτοιο ὥστε $B \wedge H \models D$.

Ἕνας τυπικὸς ἀλγόριθμος ΕΛΠ κατασκευάζει τὸ πρόγραμμα H ἐξετάζοντας μία λογικὴ πρόταση τὴν φορά, ἀκολουθώντας τὴν στρατηγικὴ *αὐξανόμενης κάλυψης* (incremental cover) ὅπου ἡ κατασκευὴ τῆς κάθε πρότασης ξεκινᾶ μία καινούργια, ἀνεξάρτητη ἀπὸ τὶς προηγούμενες ἀναζήτηση διαμέσου τοῦ χώρου ὅλων τῶν δυνάμει προτάσεων. Ἡ ἀναζήτηση διενεργεῖται μὲ τὴν βοήθεια ἑνὸς *τελεστῆ ἐξειδίκευσης* ὁ ὁποῖος ὁρίζει μία μερικὴ ταξινόμηση κατὰ μῆκος τοῦ ἄξονα γενικοῦ-εἰδικοῦ καὶ μᾶς ἐπιτρέπει νὰ διατρέξουμε τὸν χῶρο κινούμενοι ἀπὸ τὸ τὶς πιὸ γενικὲς προτάσεις πρὸς τὶς πιὸ εἰδικές. Ἐναλλακτικά, μποροῦμε νὰ νὰ χρησιμοποιήσουμε ἕναν *τελεστῆ γενίκευσης* καὶ νὰ κινηθοῦμε ἀπὸ τὸ εἰδικὸ πρὸς τὸ γενικό. Τὸ πιὸ γενικὸ ἄκρο τοῦ χώρου εἶναι ἡ κενή, μή-συνεπὴς πρόταση $\square$. Τὸ σύνορο στὴν εἰδικὴ πλευρὰ τοὺ χώρου τίθεται ἀπὸ τὴν ἐλάχιστη γενίκευση ἑνὸς θετικοῦ παραδείγματος (*bottom clause*). Ὁ Progol εἶναι ἕνα παράδειγμα τέτοιου ἀλγόριθμου ΕΛΠ καὶ τὸ Aleph ἕνα σύστημα ΕΛΠ ποὺ ὑλοποιεῖ, μεταξὺ ἄλλων, τὸν Progol. (Βλ. δεύτερο κε-

φάλαιο γιὰ περισσότερες λεπτομέρειες.)

Τὸ δεύτερο κεφάλαιο κλείνει μὲ τὴν διερεύνηση τῆς ἐπιλογῆς τοῦ ΕΛΠ γιὰ τὴν ἐκτέλεση γλωσσολογικῶν πειραμάτων. Ὁ ΕΛΠ ὑποφέρει ἀπὸ τὰ ἴδια προβλήματα καὶ ἀπολαμβάνει τῶν ἴδιων πλεονεκτημάτων ὅπως καὶ κάθε μορφὴ συμβολικοῦ λογισμοῦ: ἔχει ὑψηλὲς ὑπολογιστικὲς ἀπαιτήσεις (τόσο γιὰ τὴν κατασκευή, ὅσο καὶ γιὰ τὴν ἐφαρμογὴ θεωριῶν), καὶ ἔχει ἐγγενεῖς δυσκολίες στὸν χειρισμὸ ἀριθμητικῶν δεδομένων. Ἀπὸ τὴν ἄλλη ὅμως πλευρά, ὁ ΕΛΠ, καὶ ἡ Τυπικὴ Λογικὴ γενικότερα, εἶναι ἐλκυστικὲς λύσεις γιὰ τὸν χειρισμὸ συμβολικῶν δεδομένων, ὅπως ἐπίσης καὶ ὅταν ἡ ἴδια ἡ «διαδικασία ἐπίλυσης» (ἡ ἴδια ἡ κατασκευασθεῖσα θεωρία) εἶναι ἐνδιαφέρουσα καὶ ὄχι μόνο ἡ «λύση» (τὸ ἀποτέλεσμα τῆς ἐφαρμογῆς τῆς θεωρίας).

Ὅπως ἤδη ἀναφέραμε, ὁ ΕΛΠ εἶναι ὑπολογιστικὰ ἀπαιτητικός, γεγονὸς τὸ ὁποῖο τὸν καθιστᾶ ἐλκυστικὸ στόχο παραλληλοποίησης. Τὸ τρίτο κεφάλαιο ξεκινᾶ μὲ μιὰ σύντομη εἰσαγωγὴ στὴν Message Passing Interface (MPI). Ἡ MPI εἶναι μία τυποποίηση τῆς Ἐπαφῆς Πραγραμματιστῆ Ἐφαρμογῶν (Application Programmers Interface, API) βιβλιοθηκῶν ρουτινῶν περάσματος μηνυμάτων γιὰ παράλληλο καὶ κατανεμημένο προγραμματισμό. Ἀκολουθεῖ μία περιγραφὴ τοῦ Aleph/MPI, μίας τροποποίησης τοῦ Aleph ποὺ κατανέμει τὰ δεδομένα καὶ ἐπαληθεύει τὴν ὑπόθεση παράλληλα. Ἡ τροποποίηση αὐτὴ ἀναπτύχθηκε κατὰ τὴν διάρκεια τῆς ἐργασίας ποὺ περιγράφεται ἐδὼ καὶ βασίζεται σὲ μία ἐπέκταση τοῦ YAP μὲ API γιὰ βιβλιοθῆκες ποὺ ἀκολουθοῦν τὴν MPI. Ὁ YAP εἶναι ἕνα σύστημα Λογικοῦ Προγραμματισμοῦ σὲ Prolog καὶ ἡ ἐπέκταση YAP/MPI εἶναι ἐπίσης προϊὸν αὐτῆς ἐδὼ τῆς ἐργασίας. Τὸ κεφάλαιο κλείνει μὲ δοκιμὲς τοῦ συστήματος Aleph/MPI καὶ διερεύνηση πιθανῶν βελτιώσεων καὶ ἐφαρμογῶν του.

Τὰ ἐπόμενα δύο κεφάλαια περιγράφουν τὴν ἐφαρμογὴ τοῦ ΕΛΠ σὲ δύο γλωσσολογικὰ ζητήματα: τὴν *ἀβαθὴ συντακτικὴ ἀνάλυση* καὶ τὴν *φωνοτακτική*. Πιὸ συγκεκριμένα, τὸ τέταρτο κεφάλαιο πραγματεύεται ἕνα πείραμα ἐπαγωγῆς ἑνὸς *κερματιστῆ κατὰ ΟΣ βάσης* (BaseNP chunker) γιὰ τὴν ἀγγλική. Ὁ κερματισμὸς κατὰ ΟΣ βάσης εἶναι ἕνα εἶδος ἀβαθοῦς συντακτικῆς ἀνάλυσης τὸ ὁποῖο ἀναγνωρίζει μόνο τὰ ὀνοματικὰ σύνολα (ΟΣ) ποὺ βρίσκονται στὸ κατώτερο ἐπίπεδο τοῦ δέντρου ποὺ δίνει τὴν συντακτικὴ δομὴ μιᾶς φράσης. Γιὰ παράδειγμα, ἂν ἡ πλήρης ἀνάλυση τοῦ ΟΣ *confidence in the pound* εἶναι:

(1) [NP [N1 Confidence] [PP in [N1 the pound]]]

τότε ὁ κερματισμὸς ποὺ μόλις περιγράψαμε εἶναι:

(2) [Confidence] in [the pound]

Ὁ κερματιστὴς δέχεται ὡς εἴσοδο κείμενο ὅπου κάθε λέξη συνοδεύεται ἀπὸ

τὸ μέρος τοῦ λόγου ὅπου ἀνήκει ἀλλὰ καὶ ὅποιες ἄλλες πληροφορίες προκύπτουν ἀπὸ τὴν μορφολογία της. Ὁ κερματισμὸς συνήθως διεκπεραιώνεται ἀπὸ αὐτόματα πεπερασμένων καταστάσεων μὲ ἕνα μόνο πέρασμα τοῦ κειμένου.

Στὸ πείραμα ποὺ περιγράφουμε στὸ τέταρτο κεφάλαιο ὁ κερματιστὴς σημαδεύει κάθε λέξη ὡς «μέσα» (I) ἢ «ἔξω» (O), ἀνάλογα μὲ τὸ ἄν ἀποτελεῖ μέρος ἑνὸς ΟΣ βάσης ἢ ὄχι. Κατ' αὐτὸν τὸν τρόπο, στὸ παράδειγμά μας θὰ λαμβάναμε:

(3) Confidence/I in/O the/I pound/I

ὡς ἔξοδο. Ἡ θεωρία ποὺ θέλουμε νὰ ἐπάγουμε ὁρίζει τὴν σχέση ἀνάμεσα σὲ μία λέξη καὶ τὰ συμφραζόμενά της καὶ τὸ σημάδι I/O ποὺ πρέπει νὰ λάβει. Μόνο τὰ ἀριστερὰ συμφραζόμενα φέρουν ἤδη τὸ σημάδι I/O, ὁπότε μία φράση μπορεῖ νὰ ἀναλυθεῖ μὲ ἕνα μόνο πέρασμα ἀπὸ τὰ ἀριστερὰ πρὸς τὰ δεξιά. Τὰ δεδομένα προέρχονται ἀπὸ τὸ Penn TreeBank, μία συλογὴ συντακτικὰ ἀναλυμένων ἀγγλικῶν προτάσεων.

Οἱ ἐπιδόσεις τοῦ ἀναλυτῆ αὐτοῦ εἶναι κατώτερες τῶν ἐπιδόσεων στοχαστικῶν μοντέλων. Ἐπίσης, ἡ θεωρία δὲν εἶναι τόσο εὐκολα ἀναγνώσιμη καὶ κατανοήσιμη ὅσο θὰ περιμέναμε ἀπὸ ἕνα λογικὸ πρόγραμμα. Αὐτὸ κυρίως καταλογίζεται στὸ πειραματικὸ στήσιμο ποὺ ἀναγκάζει ἕνα συντακτικὸ φαινόμενο νὰ ταιριάξει στὸ καλούπι ἑνὸς τοπικοῦ φαινομένου.

Τὸ πέμπτο κεφάλαιο περιγράφει τὴν ἐφαρμογὴ τοῦ ΕΛΠ στὴν φωνολογία, καὶ πιὸ συγκεκριμένα τήν *φωνοτακτική:* τὸ σύνολο τῶν κανόνων ποὺ ὁρίζουν ποιοὶ συνδυασμοὶ ἤχων εἶναι ἐπιτρεπτοὶ σὲ μία γλῶσσα καὶ ποιοὶ ὄχι. Μὲ ἄλλα λόγια, τὸ φωνοτακτικὸ μοντέλο μιᾶς γλώσσας ξεχωρίζει τὶς λέξεις ποὺ δὲν ἀπαντῶνται στὸ λεξικὸ σὲ δύο κατηγορίες: τὰ *συστηματικὰ κενὰ* ποὺ ἀπαρτίζονται ἀπὸ συνδυασμοὺς ἤχων ποὺ δὲν θὰ μπορούσαν νὰ εἶναι λέξεις, στὰ ἑλληνικὰ π.χ. /ɣrfo/· καὶ τὰ *τυχαῖα κενὰ* ποὺ θὰ μπορούσαν ἀλλὰ τυχαίνει ἁπλῶς νὰ μὴν εἶναι, π.χ. /ˈɣrəfo/.

Τὸ πειραματικὸ στήσιμο παραμένει σὲ γενικὲς ὅμοιο μὲ αὐτὸ τοῦ προηγούμενου πειράματος: τὸ ζητούμενο εἶναι ἕνας ἀναγνώστης ὁ ὁποῖος μὲ ἕνα πέρασμα ἀπὸ τὸ φωνῆεν πρὸς τὰ ἔξω θὰ ἀναγνωρίζει τὶς ἐπιτρεπτὲς συλλαβὲς καὶ θὰ ἀπορρίπτει αὐτὲς ποὺ παραβιάζουν κάποιον φωνοτακτικὸ κανόνα. Τὸ πείραμα περιορίζεται στὴν ἀναγνώριση μεμονομένων συλλαβῶν, καθὼς ὑποθέτουμε πὼς τὸ συνολικὸ φωνοτακτικὸ μοντέλο εἶναι ὁ συνδυασμὸς τοῦ συλλαβικοῦ φωνοτακτικοῦ μοντέλου καὶ τῶν κανόνων σύνθεσης λέξεων ἀπὸ τὶς ἐπιμέρους συλλαβές. Μὲ ἄλλα λόγια, ὑποθέτουμε πὼς ὅλοι οἱ συνδυασμοὶ ἐπιτρεπτῶν συλλαβῶν μποροῦν νὰ συνθέσουν ἐπιτρεπτὲς λέξεις.

Τὸ ζητούμενο εἶναι ἡ σχέση ποὺ ὁρίζει ποιὰ σύμφωνα μποροῦν νὰ ἀποτελέσουν ἐπιτρεπτὸ πρόθεμα ἢ ἐπίθεμα σὲ μία μερικῶς ἀναγνωρισμένη συλλαβή.

(Χωρὶς νὰ σημαίνει πὼς πρέπει ἀπαραίτητα νὰ θεωρήσουμε ὅλα τὰ μέρη ἐπιτρεπτῶν συλλαβῶν ἐπίσης ἐπιτρεπτά.) Τὰ δεδομένα προέρχονται ἀπὸ τὴν Λεξιλογικὴ Βάση Δεδομένων CELEX, καὶ συγκεκριμένα ἀπὸ τὸ ὁλλανδικὸ τμῆμα. Τὸ πείραμα ἐπαναλήφθηκε χρησιμοποιώντας δύο διαφορετικὰ φωνολογικὰ μοντέλα γιὰ τὴν ἀναπαράσταση τῶν συμφώνων, τὸ ἕνα ἀπὸ τὰ ὁποῖα εἶναι ἕνα μοντέλο τῆς ὁλλανδικῆς φωνολογίας καὶ περιέχει πολὺ περισσότερη πληροφορία ἀπ' ὅτι τὸ ἄλλο. Τὰ ἀποτελέσματα συγκρίνονται μεταξὺ τους καὶ μὲ ἕνα χειροποίητο φωνοτακτικὸ μοντέλο τῶν ὁλλανδικῶν. Τὸ συμπέρασμα ποὺ προκύπτει ἀπὸ τὴν σύγκριση εἶναι πὼς ὁ ἀλγόριθμος ΕΛΠ ἐπωφελήθηκε ἀπὸ τὴν παραπάνω πληροφορία τοῦ προσαρμοσμένου στὰ ὁλλανδικὰ μοντέλου καὶ κατασκεύασε μιὰ θεωρία ποὺ καὶ συνοπτικότερη εἶναι καὶ καλύτερες ἐπιδόσεις ἐπιτυγχάνει ἀπ' ὅτι ὅταν βασίστηκε στὸ γενικὸ φωνολογικὸ μοντέλο.

Τὸ ἔκτο καὶ τελευταῖο κεφάλαιο συνοψίζει καὶ ἀναλύει τὰ ἐπιμέρους συμπεράσματα τῶν προηγούμενων κεφαλαίων, ἀλλὰ καὶ ἐξαγάγει συμπεράσματα ἀπὸ τὸ σύνολο τῆς διατριβῆς. Εἰδικότερα, ἀνακεφαλαιώνει τὰ προβλήματα ποὺ ἀντιμετωπίζει ὁ ΕΛΠ στὸν κερματισμὸ, κυρίως τὴν ἔλλειψη ἀπριορικῆς γνώσης καὶ τὴν μὴ ἀναγνωσιμότητα τῆς θεωρίας. Σὲ σχέση μὲ τὰ φωνοτακτικὰ πειράματα, συμπεραίνεται πὼς ἡ ἐπιτυχία τῶν τελευταίων ὀφείλεται στὸ ὅτι ἡ φωνολογία εἶναι ἕνα τοπικὸ φαινόμενο ποὺ ταιριάζει καλύτερα στὸ πειραματικὸ στήσιμο ποὺ υἱοθετήθηκε, ἀλλὰ καὶ γιατὶ ἔγινε καλύτερη χρήση τοῦ μηχανισμοῦ ἐκ τῶν προτέρων γνώσης ποὺ παρέχει ὁ ΕΛΠ.

Bibliography

[1] Steven Abney. Parsing by chunks. In Robert Berwick, Steven Abney, and Carol Tenny, editors, *Principle-Based Parsing*. Kluwer Academic Publishers, Dordrecht, 1991. URL `http://www.vinartus.net/spa/90e.pdf`.

[2] International Phonetic Association. *IPA Chart*. `http://www.arts.gla.ac.uk/IPA/ipachart.html`, Last update: 1996.

[3] Simon Blackburn. *The Oxford Dictionary of Philosophy*. Oxford University Press, 1996. URL `http://www.oxfordreference.com/BOOK_SEARCH.html?book=t98&subject=s22`.

[4] Geert Booij. *The Phonology of Dutch*. The Phonology of the World's Languages. Clarendon Press, Oxford, 1995.

[5] Henrik Boström and Lars Asker. Combining divide-and-conquer and separate-and-conquer for efficient and effective rule induction. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, number 1634 in Lecture Notes in AI Series. Springer, 1999. URL `http://www.dsv.su.se/~henke/PAPERS/ilp99.ps`.

[6] Henrik Boström and Peter Idestam-Almquist. Induction of logic programs by example-guided unfolding. *Journal of Logic Programming*, 40(2–3):159–183, 1999. URL `http://www.dsv.su.se/~henke/PAPERS/jlp.ps`.

[7] Thorsten Brants. Cascaded Markov models. In *Proceedings of the Ninth Conference of the European Chapter of the ACL*, 1999.

[8] Eric Brill. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–66, 1995.

[9] J. S. Bruner, J. J. Goodnow, and G. A. Austin. *A Study of Thinking*. Wiley, New York, 1956.

[10] Rui Camacho. The use of background knowledge in Inductive Logic Programming. Technical report, Oxford University, March 1994. URL `http://www.liacc.up.pt/~tau/publications/qualifying.ps.gz`.

[11] Bojan Cestnik. Estimating probabilities: A crucial task in machine learning. In *European Conference on Artificial Intelligence*, pages 147–149, 1990.

[12] Noam Chomsky. *Syntactic Structures*. Mouton, The Hague, 1957. first mention of trees and such.

[13] Alonzo Church. *Introduction to Mathematical Logic*. Princeton University Press, Princeton, NJ, 1958.

[14] Peter Clark and Robin Boswell. Rule induction with CN2: Some recent improvements. In Yves Kodratoff, editor, *Machine Learning – Proceedings of the Fifth European Conference*, pages 151–163, Berlin, 1991. Springer-Verlag.

[15] Peter Clark and Tim Niblett. Induction in noisy domains. In Ivan Bratko and Nada Lavrač, editors, *Progress in Machine Learning: Proceedings of the 2nd European Working Session on Learning*, pages 11–30, Wilmslow, U.K., 1987.

[16] Peter Clark and Tim Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4):261–83, 1989.

[17] William W. Cohen. Fast effective rule induction. In Prieditis and Russell [63]. URL `http://www-2.cs.cmu.edu/~wcohen/postscript/ml-95-ripper.ps`.

[18] William W. Cohen and Haym Hirsh, editors. *Machine Learning: Proceedings of the Eleventh International Conference*, 1994. Morgan Kaufmann.

[19] Vítor Santos Costa. *The YAP Prolog System*. `http://yap.sourceforge.net/`.

[20] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications. Wiley, New York, 1991.

[21] Luc De Raedt and Luc Dehaspe. Clausal discovery. Technical Report CW 238, Department of Computing Science, K.U.Leuven, Leuven, 1996. URL `ftp://ftp.cs.kuleuven.ac.be/pub/logic-prgm/ilp/claudien/papers/claudien_paper.ps.gz`.

[22] Luc Dehaspe and Luc De Raedt. Parallel inductive logic programming. In *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, 1995.

[23] Sašo Džeroski and Ivan Bratko. Handling noise in Inductive Logic Programming. In *Proceedings of the Second International Workshop on Inductive Logic Programming*, Japan, 1992. Institute for New Generation Computing.

[24] MPI Forum. *MPI: A Message-Passing Interface Standard.* `http://www.mpi-forum.org/docs/docs.html`, June 1995.

[25] MPI Forum. *MPI-2: Extensions to the Message Passing Interface.* `http://www.mpi-forum.org/docs/docs.html`, July 1997.

[26] Johannes Fürnkranz and Gerhard Widmer. Incremental reduced error pruning. In Cohen and Hirsh [18].

[27] William D. Gropp and Ewing Lusk. *User's Guide for MPICH, a Portable Implementation of MPI.* Mathematics and Computer Science Division, Argonne National Laboratory, 1996. URL `http://www-unix.mcs.anl.gov/mpi/mpich/docs.html`. ANL-96/6.

[28] Zellig S. Harris. Co-occurence and transformation in linguistic structure. *Language*, 33(3):283–340, 1957.

[29] Earl B. Hunt, Janet Marin, and Philip T. Stone. *Experiments in Induction.* Academic Press, New York, 1966.

[30] ILK Research Group, Tilburg and CNTS Research Group, Antwerp. *TiMBL: Tilburg Memory Based Learner.* `http://ilk.kub.nl/software.html`, Last update: July 2003.

[31] W. Stanley Jevons. *Elementary Lessons in Logic: Deductive and Inductive: with copious questions and examples and a vocabulary of logical terms.* Macmillan, London, 1870.

[32] Igor Kononenko, Ivan Bratko, and Egidija Roskar. Experiments in automatic learning of medical diagnostic rules. Technical report, Jožef Stefan Instutute, Ljubljana, Yugoslavia, 1984.

[33] Stasinos Th. Konstantopoulos. NP chunking using ILP. In Paola Monachesi, editor, *Proceedings of Computational Linguistics in the Netherlands 1999*, pages 109–116, Utrecht, 2000. Utrecht Institute of

Linguistics OTS. URL `ftp://ftp.let.rug.nl/pub/konstant/Docs/clin1999.ps.bz2`.

[34] Stasinos Th. Konstantopoulos. Learning phonotactics using ILP. In Kristina Striegnitz, editor, *Proc. of the Sixth ESSLLI Student Session*, pages 148–158, Helsinki, 2001. URL `ftp://ftp.let.rug.nl/pub/konstant/Docs/esslli01.ps.bz2`.

[35] Stasinos Th. Konstantopoulos. Learning phonotactics using ILP. *Special Issue of the WEB-SLS On-line Journal: The Language Sections of the ESSLLI-01 Student Session*, 2002. URL `http://hltheses.elsnet.org/eventpapers/essliproceed.htm`.

[36] Stasinos Th. Konstantopoulos. BaseNP chunking using ILP. In *Computational Linguistics in the Netherlands 2002*, 2003. Proceedings to be published in December 2003.

[37] Stasinos Th. Konstantopoulos. A data-parallel version of Aleph. In Rui Camacho and Ashwin Srinivasan, editors, *Proc. of the Workshop on Parallel and Distributed Computing for Machine Learning Workshop, ECML/PKDD 2003*, 2003. URL `ftp://ftp.let.rug.nl/pub/konstant/Docs/ecml03.pdf`.

[38] Taku Kudoh and Yuji Matsumoto. Chunking with support vector machines. In *Proceedings of the Second Meeting of North American Chapter of Association for Computational Linguistics, Pittsburgh, USA*, pages 192–199, June 2001.

[39] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1997.

[40] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: the penn treebank. *Computational Linguistics*, 19, 1993. URL `http://morph.ldc.upenn.edu/Catalog/docs/treebank2/cl93.html`.

[41] P. H. Matthews, editor. *The Concise Oxford Dictionary of Linguistics*. Oxford University Press, 1997. URL `http://www.oxfordreference.com/views/BOOK_SEARCH.html?book=t36`.

[42] James McCosh. *The Scottish Philosophy*. Macmillan, London, 1875. URL `http://www.socsci.mcmaster.ca/~econ/ugcm/3ll3/mccosh/scottishphilosophy.pdf`.

[43] Ryszard Michalski. On the quasi-minimal solution of the general covering problem. In *Proceedings of the Fifth International Symposium on Information Processing*, volume A3, pages 125–128, Bled, Yugoslavia, 1969.

[44] John Stuart Mill. Book II: Reasoning. In *A System of Logic* Mill [46]. First published in 1843.

[45] John Stuart Mill. Book III: Induction. In *A System of Logic* Mill [46]. First published in 1843.

[46] John Stuart Mill. *A System of Logic*. Longmans, Green, and Co., London, tenth edition, 1879. First published in 1843.

[47] Tom Mitchell. *Machine Learning*, chapter 10, Learning Sets of Rules. McGraw Hill, second edition, 1997.

[48] Ernest A. Moody. *The Logic of William of Ockham*. Sheed & Ward, London, 1935.

[49] Stephen Muggleton. Duce, an oracle based approach to constructive induction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 187–292. Kaufmann, 1987.

[50] Stephen Muggleton. A strategy for constructing new predicates in first order logic. In D. Sleeman, editor, *Proceedings of the 3rd European Working Session on Learning*, pages 123–130. Pitman, 1988. URL `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/ewsl88.ps.gz`.

[51] Stephen Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995. URL `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/InvEnt.ps.gz`.

[52] Stephen Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the 5th International Conference on Machine Learning*, pages 339–352. Kaufmann, 1988.

[53] Stephen Muggleton and Luc De Raedt. Inductive Logic Programming: Theory and methods. *Journal of Logic Programming*, 19(20):629–679, 1994. URL `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/lpj.ps.gz`. Updated version of technical report CW 178, May 1993, Department of Computing Science, K.U. Leuven.

[54] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381, 1990. URL `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/alt90.ps`.

[55] Stephen H. Muggleton. Learning from positive data. In *Proceedings of the Sixth International Workshop on Inductive Logic Programming, LNAI 1314*, pages 358–376, Berlin, 1996. Springer-Verlag.

[56] Marcia Muñoz, Vasin Punyakanok, Dan Roth, and Dav Zimak. A learning approach to shallow parsing. In *Proceedings of EMNLP-WVLC '99*. Association for Computational Linguistics, 1999. URL `http://l2r.cs.uiuc.edu/~danr/Papers/emnlp99.ps.gz`.

[57] Hayato Ohwada, Hiroyuki Nishiyama, and Fumio Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. In James Cussens and Alan M. Frisch, editors, *Proceedings of the 10th International Workshop on Inductive Logic Programming (ILP 2000)*, volume 1866 of *Lecture Notes in Computer Science*, pages 165–173. Springer Verlag, 2000. URL `http://link.springer.de/link/service/series/0558/bibs/1866/18660165.htm`.

[58] Peter S. Pacheco. *A User's Guide to MPI*, March 1998. URL `ftp://math.usfca.edu/pub/MPI/`.

[59] Richard Piepenbrock. *CELEX, The Dutch Centre For Lexical Information*. `http://www.kun.nl/celex/`, Last update: February 2001.

[60] Gordon D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1969.

[61] Gordon D. Plotkin. A further note on inductive generalization. In D. Michie, N. L. Collins, and E. Dale, editors, *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, 1971.

[62] Karl Popper. *The Logic of Scientific Discovery*. Hutchinson and Co., 1959.

[63] Armand Prieditis and Stuart J. Russell, editors. *Machine Learning: Proceedings of the Twelfth International Conference*, 1995. Morgan Kaufmann.

[64] J. Ross Quinlan. Discovering rules by induction from large collections of examples. In Donald Michie, editor, *Expert Systems in the Micro Electronic Age*, pages 168–201. Edinburgh University Press, Edinburgh, 1979.

[65] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1): 81–106, 1986.

[66] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[67] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1992.

[68] J. Ross Quinlan. MDL and categorical theories (continued). In Prieditis and Russell [63].

[69] Lance A. Ramshaw and Mitchell P. Marcus. Text chunking using transformation-based learning. In *Proceedings of the Third ACL Workshop on Very Large Corpora*. Association for Computational Linguistics, 1995.

[70] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[71] Ricardo Jorge Rocha. *On Applying Or-Parallelism and Tabling to Logic Programs*. PhD thesis, Dept. of Computer Science, University of Porto, November 2001. URL `http://www.ncc.up.pt/~ricroc/research/thesis_phd.ps.gz`.

[72] Beatrice Santorini. Part-of-speech tagging guidelines for the Penn Treebank project. Technical report, The Penn Treebank Project, 1990. URL `ftp://ftp.cis.upenn.edu/pub/treebank/doc/tagguide.ps.gz`. 3rd Revision, 2nd Printing (Feb. 1995).

[73] Cullen Schaffer. A conservation law for generalization performance. In Cohen and Hirsh [18].

[74] Claude E. Shannon. The mathematical theory of communication. *Bell Systems Technical Journal*, 27:379–423, 623–656, July, October 1948.

[75] Stewart Shapiro. Book II: History. In *Thinking about Mathematics: the Philosophy of Mathematics*. Oxford University Press, 2000.

[76] Stuart Shieber. *An Introduction to Unification Approaches To Grammar*. Number 4 in CSLI Lecture Notes. Centre for the Study of Language and Information, 1986.

[77] David B. Skillicorn and Yu Wang. Parallel and sequential algorithms for data mining using inductive logic. *Knowledge and Information Systems*, 3(4):405–421, 2001. URL `http://link.springer.de/link/service/journals/10115/bibs/1003004/10030405.htm`.

[78] John Skorupski, editor. *The Cambridge Companion to Mill*. Cambridge University Press, Cambridge, 1998.

[79] Ashwin Srinivasan. *The Aleph Manual*. `http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`, Last update: Nov 21, 2002.

[80] Ivilin Stoianov. *Connectionist Lexical Processing*. PhD thesis, Rijksuniversiteit Groningen, 2001.

[81] Ivilin Stoianov and John Nerbonne. Exploring phonotactics with Simple Recurrent Networks. In van Eynde, Schuurman, and Schelkens, editors, *Proceedings of Computational Linguistics in the Netherlands 1998*, 1999.

[82] Erik F. Tjong Kim Sang and Sabine Buchholz. Introduction to the CoNLL-2000 shared task: Chunking. In Claire Cardie, Walter Daelemans, Claire Nédellec, and Erik F. Tjong Kim Sang, editors, *Proceedings of the Fourth Conference on Computational Language Learning, Lisbon*, pages 127–132, 2000. URL `http://lcg-www.uia.ac.be/lcg/ps/tks.conll2000.ps.gz`.

[83] Erik F. Tjong Kim Sang and John Nerbonne. Learning the logic of simple phonotactics. In James Cussens and Sašo Džeroski, editors, *Learning Language in Logic*, volume 1925 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 2000.

[84] Erik F. Tjong Kim Sang and Jorn Veenstra. Representing text chunks. In *Proceedings of the Ninth Conference of the European Chapter of the ACL*, 1999.

[85] Hugh Tredennick, editor. *Prior Analytics*, volume I of *Aristotle in Twenty-Three Volumes*. William Heinemann, London and Harvard University Press, Cambridge, Massachusetts, 1973.

[86] Hugh Tredennick, editor. *Posterior Analytics*, volume II of *Aristotle in Twenty-Three Volumes*. William Heinemann, London and Harvard University Press, Cambridge, Massachusetts, 1976.

[87] Harry van der Hulst. *Syllable Structure and Stress in Dutch*. PhD thesis, Rijksuniversiteit Leiden, 1984.

[88] William Whewell. *On the Philosophy of Discovery, Chapters Historical and Critical: Including the Completion of the Philosophy of Inductive Science*. Parker, London, 1860.

[89] Edward Zalta, editor. *Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, Stanford University, `http://plato.stanford.edu/`, Last update: 2003.

Groningen Dissertations in Linguistics

1. Henriëtte de Swart (1991). *Adverbs of Quantification: A Generalized Quantifier Approach.*
2. Eric Hoekstra (1991). *Licensing Conditions on Phrase Structure.*
3. Dicky Gilbers (1992). *Phonological Networks. A Theory of Segment Representation.*
4. Helen de Hoop (1992). *Case Configuration and Noun Phrase Interpretation.*
5. Gosse Bouma (1993). *Nonmonotonicity and Categorial Unification Grammar.*
6. Peter I. Blok (1993). *The Interpretation of Focus.*
7. Roelien Bastiaanse (1993). *Studies in Aphasia.*
8. Bert Bos (1993). *Rapid User Interface Development with the Script Language Gist.*
9. Wim Kosmeijer (1993). *Barriers and Licensing.*
10. Jan-Wouter Zwart (1993). *Dutch Syntax: A Minimalist Approach.*
11. Mark Kas (1993). *Essays on Boolean Functions and Negative Polarity.*
12. Ton van der Wouden (1994). *Negative Contexts.*
13. Joop Houtman (1994). *Coordination and Constituency: A Study in Categorial Grammar.*
14. Petra Hendriks (1995). *Comparatives and Categorial Grammar.*
15. Maarten de Wind (1995). *Inversion in French.*
16. Jelly Julia de Jong (1996). *The Case of Bound Pronouns in Peripheral Romance.*
17. Sjoukje van der Wal (1996). *Negative Polarity Items and Negation: Tandem Acquisition.*
18. Anastasia Giannakidou (1997). *The Landscape of Polarity Items.*
19. Karen Lattewitz (1997). *Adjacency in Dutch and German.*
20. Edith Kaan (1997). *Processing Subject-Object Ambiguities in Dutch.*
21. Henny Klein (1997). *Adverbs of Degree in Dutch.*
22. Leonie Bosveld-de Smet (1998). *On Mass and Plural Quantification: The case of French 'des'/'du'-NPs.*
23. Rita Landeweerd (1998). *Discourse semantics of perspective and temporal structure.*
24. Mettina Veenstra (1998). *Formalizing the Minimalist Program.*
25. Roel Jonkers (1998). *Comprehension and Production of Verbs in aphasic Speakers.*

26. Erik Tjong Kim Sang (1998). *Machine Learning of Phonotactics.*
27. Paulien Rijkhoek (1998). *On Degree Phrases and Result Clauses.*
28. Jan de Jong (1999). *Specific Language Impairment in Dutch: Inflectional Morphology and Argument Structure.*
29. H. Wee (1999). *Definite Focus.*
30. E-H. Lee (2000). *Dynamic and Stative Information in Temporal Reasoning: Korean tense and aspect in discourse.*
31. Ivilin P. Stoianov (2001). *Connectionist Lexical Processing.*
32. Klarien van der Linde (2001). *Sonority substitutions.*
33. Monique Lamers (2001). *Sentence processing: using syntactic, semantic, and thematic information.*
34. Shalom Zuckerman (2001). *The Acquisition of "Optimal" Movement.*
35. Rob Koeling (2001). *Dialogue-Based Disambiguation: Using Dialogue Status to Improve Speech Understanding*
36. Esther Ruigendijk (2002). *Case assignment in agrammatism: a cross-linguistic study*
37. Anthony J. Mullen (2002). *An Investigation into Compositional Features and Feature Merging for Maximum Entropy-Based Parse Selection.*
38. Nanette Bienfait (2002). *Grammatica-onderwijs aan allochtone jongeren.*
39. Dirk-Bart den Ouden (2002). *Phonology in Aphasia: Syllables and Segments in Level-Specified Deficits.*
40. Rienk Withaar (2002). *The Role of the Phonological Loop in Sentence Comprehension.*
41. Kim Sauter (2002). *Transfer and Access to Universal Grammar in Adult Second Language Acquisition.*
42. Laura Sabourin (2003). *Grammatical Gender and Second Language Processing: An ERP Study.*
43. Hein van Scie (2003). *Visual Semantics.*
44. Lilia Schürcks-Grozeva (2003). *Binding and Bulgarian.*

GRODIL, secretary Department of General Linguistics
P.O. Box 716
9700 AS Groningen
The Netherlands